Calhoun
Institutional Archive of the Naval Postgraduate School

Calhoun: The NPS Institutional Archive

DSpace Repository

Theses and Dissertations | 1. Thesis and Dissertation Collection, all items

1993

NPSNET : integration of distributed interactive simulation (DIS) protocol for communication architecture and information interchange

Zeswitz, Steven Randall.

Monterey, California. Naval Postgraduate School

http://hdl.handle.net/10945/40018

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun

http://www.nps.edu/library

Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

NPSNET: INTEGRATION OF DISTRIBUTED INTERACTIVE SIMULATION (DIS) PROTOCOL FOR COMMUNICATION ARCHITECTURE AND INFORMATION INTERCHANGE

by

Steven Randall Zeswitz

September 1993

Thesis Advisor: Dr. David R. Pratt
Second Reader: Dr. Gilbert M. Lundy

Approved for public release; distribution is unlimited

Thesis
Z3433

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time reviewing instructions, searching existing data sources gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1993	Master's Thesis

4. TITLE AND SUBTITLE
NPSNET: Integration of Distributed Interactive Simulation (DIS) Protocol for Communication Architecture and Information Interchange

5. FUNDING NUMBERS

6. AUTHOR(S)
Zeswitz, Steven Randall

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/ MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/ MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release;
distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Computer Science Department at the Naval Postgraduate School in Monterey, California has developed a low-cost real-time interactive network based simulation system, known as NPSNET, that uses Silicon Graphics workstations. NPSNET has used non-standard protocols which constrains its participation in distributed simulation. DIS specifies standard protocols and is emerging as the international standard for distributed simulation.

This research focused on the development of a robust, high-performance implementation of the DIS Version 2.0.3 protocol to support graphic simulation systems (e.g. NPSNET). The challenge was to comply with the standard and minimize network latency thereby maintaining the time and space coherence of distributed simulations. The resulting DIS Network Library consists of an application program interface (API) to low level network routines, a host of network utilities, and a network harness that takes advantage of multiprocessor workstations.

The library was successfully tested on our local network and two configurations of a T-1 based internet, the Defense Simulation Internet (DSI), with the Air Force Institute of Technology and Advanced Research Projects Agency. The testing confirmed that the semantics and syntax of the DIS protocol is properly implemented and the latency incurred by the network does not adversely effect the simulation application.

14. SUBJECT TERMS	15. NUMBER OF PAGES
DIS, distributed, interactive, simulation, networking, distributed system, UDP/IP, sockets, UNIX, IPC, client/server model,	80
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

Approved for public release; distribution is unlimited

NPSNET: INTEGRATION OF DISTRIBUTED INTERACTIVE SIMULATION (DIS) PROTOCOL FOR COMMUNICATION ARCHITECTURE AND INFORMATION INTERCHANGE

by
Steven R. Zeswitz
Captain, United States Marine Corps
B.S., National University, 1987

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1993

Author: [redacted]
Steven Randall Zeswitz

Approved By: [redacted]
Dr. David R. Pratt, Thesis Advisor

[redacted]
Dr. Gilbert M. Lundy, Thesis Second Reader

[redacted]
Dr. Ted Lewis, Chairman,
Department of Computer Science

ABSTRACT

The Computer Science Department at the Naval Postgraduate School in Monterey, California has developed a low-cost real-time interactive network based simulation system, known as NPSNET, that uses Silicon Graphics workstations. NPSNET has used non-standard protocols which constrains its participation in distributed simulation. DIS specifies standard protocols and is emerging as the international standard for distributed simulation.

This research focused on the development of a robust, high-performance implementation of the DIS Version 2.0.3 protocol to support graphic simulation systems (e.g. NPSNET). The challenge was to comply with the standard and minimize network latency thereby maintaining the time and space coherence of distributed simulations. The resulting DIS Network Library consists of an application program interface (API) to low level network routines, a host of network utilities, and a network harness that takes advantage of multiprocessor workstations.

The library was successfully tested on our local network and two configurations of a T-1 based internet, the Defense Simulation Internet (DSI), with the Air Force Institute of Technology and Advanced Research Projects Agency. The testing confirmed that the semantics and syntax of the DIS protocol is properly implemented and the latency incurred by the network does not adversely effect the simulation application.

Thesis
Z34/33
c.2

TABLE OF CONTENTS

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CA 93943-5101

LIST OF FIGURES

I. INTRODUCTION

A. OBJECTIVE

The objective of this research project was to develop a robust, high-performance and efficient implementation of the Distributed Interactive Simulation (DIS) Version 2.0.3 protocol suite. DIS 2.0.3 is the latest version of this emerging international standard for distributed simulation [IST93a]. The success of this project was measured by the ability to support real-time graphic simulations in a networked environment.

B. SCOPE

The project entailed redesign of the network harness, modification of the application program interface (API) and NPSNET to comply with the DIS standard, and development of a network monitoring tool kit for troubleshooting and performance measurement. The network harness is a software architecture designed to take advantage of the multiprocessor machines in our laboratory. The architecture uses BSD 4.3 socket-based interprocess communication (IPC) to provide a clear, easily used, and well-documented network interface. NPSNET, the simulation application, is tailored with efficient mechanisms to map DIS data to NPSNET data structures. The DIS network library that we developed through this research a portable network harness for DIS applications. DIS allows real-time, three-dimensional computer simulation systems (e.g. NPSNET) to interact with other independently developed simulations (e.g. Virtual Cockpit, World Reference Model[1]) via communication networks.

Our research employed Silicon Graphics, Inc. (SGI) workstations as simulation hosts, communicating through Ethernet in the Graphics and Video Laboratory, Naval Postgraduate School (NPS), Monterey, CA and the DSI, a T-1 based internetwork. Other participating sites in this research were the Air Force Institute of Technology (AFIT),

1. The Virtual Cockpit was developed by the Air Force Institute of Technology. The World Reference Model was developed by ARPA.

Dayton, Ohio, the Simulation Center, Advanced Research Projects Agency (ARPA), Arlington, Virginia, and the demonstration booth set up by NPS, AFIT, and ARPA during the Association of Computing Machinery Special Interest Group for Graphics (ACM-SIGGRAPH) conference in Anaheim, California.

This work was a landmark for implementation of the DIS protocol suite in a functional real-time system connecting independently developed simulations on distributed local area networks via the Defense Simulation Internet (DSI), a wide area network. It was demonstrated at the ACM-SIGGRAPH conference in Anaheim, California, from 1-6 August 1993. This was the first public demonstration of the DIS protocol Version 2.0.3, showing that it is a viable protocol for nation-wide distributed interactive simulations. The demonstration allowed the general public to participate in a free-play, distributed interactive simulation. The simulated environment was populated with nine live participants in Anaheim, two live participants in Arlington, Virginia, twelve autonomous participants in Monterey, California, and one participant in Dayton, Ohio.

C. BACKGROUND

The Department of Defense and other government and civilian organizations have used simulation as a training and analysis tool for many years. The Defense Modeling and Simulation Initiative has been instituted to promote effective and efficient use of modeling and simulation in a joint environment [DoD92].

Even though future military spending cuts are expected to reduce the number and size of live training exercises, the need for realistic training will not diminish [Redd92]. Realistic training is essential for our military units to maintain their high state of combat readiness and effectiveness. Interactive simulation systems have proven to be a valuable and cost-effective method to augment live field training when the expense of live exercises is prohibitive [Pope89]. They provide a tool for the development of a smarter, more effective fighting force by augmenting operational experience and improving, expanding, and refining the thought processes of warfighters from commanders, through his/her staffs,

to the individual infantryman. Simulation technology is targeted to improve decision making and skill levels through interaction with events and familiarization with situations. Computer-based simulation training exercises permit repetition of expensive exercises. Simulations do not endanger lives, expend live ammunition, or consume other valuable and limited resources necessary for live exercises. Further, distributed simulation allows for people and/or units to interact as teams in the same virtual environment. [Thor87]

For years the United States military has conducted large-scale combined arms exercises. The exercises are expensive, but they are necessary for the successful deployment of a Joint Task Force (JTF). An effort is underway to reduce these expenses by conducting large-scale exercises in virtual environments [IST93a]. A virtual environment's most significant cost is the initial implementation of the hardware and software to model the scenario. These costs are dissipated over time as software libraries expand with standard, reusable models. These libraries are a tool that can be used to quickly configure a virtual environment with a wide range of terrain and equipment models. [Lora92]

In addition to employment in operational training, simulation is being used for rapid prototyping and analysis. The sophistication of simulations has increased to where they can be constructed in a very short period of time and quickly modified as required to provide the desired level of realism. Research in computer science and communications is contributing to the continued advancement of techniques and technologies needed to gain greater benefit from simulation as a training and analysis tool. [Bogg92][Chun92]

Advances in modeling and simulation technology have lowered the cost and increased the usefulness of sophisticated training and decision support tools that meet many military and civilian needs. The list of applications include: force planning, training and readiness, doctrinal development, combat development, logistics, operation rehearsal and planning, operation analysis, training development, and system acquisition and development. However, most simulations developed over the past two decades were designed for individuals and/or small units. [Lora92][Pope89][Redd92]

Experimentation with networked virtual environments began with the initial development of the Simulation Networking (SIMNET) project in 1983 [Pope89]. This was a DARPA project to explore large scale simulator networking. Six years later, the SIMNET system and protocols were delivered for use in DoD. SIMNET was adopted as the de facto DoD standard for distributed simulation. As SIMNET protocols were being delivered, it immediately became the baseline for a much more ambitious standard, DIS. [IST93a]

D. SUMMARY OF CHAPTERS

Chapter II provides an overview of the philosophy and components of the DIS standards. Key components of the DIS application architecture, communications architecture and protocol data units (PDUs) standards are presented. Chapter III contains a description of the NPSNET project and its achievements as a networked simulator. Chapter IV discusses the construction of the network architecture integrated with NPSNET. Chapter V presents the usage of the DIS network library. Chapter VI discusses the results of testing the network architecture locally and over long haul networks. Chapter VII contains recommended topics for continued research and concluding remarks. Appendix A is a user's guide for the functions contained in the network library. Appendix B is the source listing of the data logging program developed in conjunction with this project. The program demonstrates the use of the network routines. Appendix C contains manual pages for the user functions in the network library.

II. STANDARDS FOR DISTRIBUTED INTERACTIVE SIMULATION

This chapter provides a description of the philosophy and key components of the DIS standards, the framework in which our network library was developed. It presents the underlying goal of the standards, the DIS architecture, and the specific standards that are relevant to our implementation.

A. INTEROPERABILITY

In the computer simulation context, interoperability is the ability of a set of simulation entities to interact with an acceptable degree of fidelity [IST93a]. For example, a Marine is operating a M-1 tank simulator from a training cell at Marine Corps Base, Camp Pendleton, California. He is not alone in this virtual environment. A platoon of Light Armored Vehicle (LAV) simulators from Camp Lejeune, North Carolina, and a reserve F-18 simulation from Naval Air Station, Miramar, California, is also participating. The terrain model being used is the Delta Corridor of the Marine Corps Combat Training Center, Twenty-nine Palms, California. The tank and LAVs are high fidelity simulators procured from different vendors and the aircraft is a low fidelity simulator that was procured from a third source. The tank is facing north, the platoon of LAVs is in a line formation turning to head west 500 meters in front of the tank, and the aircraft flies over the tank heading north at 1500 feet. In an interoperable simulation, these events would be accurately represented within the fidelity capabilities of the participating simulators. Interoperability requires a sufficient level of correlation between simulations [Lora92]. In man-in-the-loop battlespace simulations, like we have just described, the correlation is measured by the perception of the operator. The key question is: Does the information display accurately reflect the spatial and temporal properties of the events being simulated? The DIS standards are being developed to facilitate an affirmative answer to this question.

B. CONCEPT

Simply stated, a distributed interactive simulation is a man-in-loop simulation in which participants interact in a shared environment from geographically dispersed sites using communication networks. A man-in-the-loop simulation is a system in which a human plays a critical role in the simulation system. The human acts as the operator, analyst, or trainee. In this type of simulation the human interacts with events and/or objects in the simulation. The shared environment is the space and time coherent "world" represented in the simulation. Geographic dispersion is limited only by the extent of the communication system supporting the simulation and the ability of the communication system to transmit events quickly enough to maintain coherence in the human perception. Distributed interactive simulation implies a degree of interoperability between different simulators. The degree of interoperability can be increased by the adoption of standards which specify a minimum level of system functionality. [Stal91]

The development of DIS standards began in 1989 as an initiative sponsored by the United States Army Simulation, Training and Instrumentation Command (STRICOM), the Advanced Research Projects Agency (ARPA), and the Defense Modeling and Simulation Office (DMSO). The University of Central Florida Institute for Simulation and Training (IST) was contracted to coordinate the development of the standards. The standardization process is an evolutionary process based on the consensus of operators, managers, technicians, analysts, and engineers from government and industry. The standards aim is to use commercial, off-the-shelf (COTS) technology and standardized protocols whenever possible. [IST93a][Lora92]

C. OBJECTIVES OF DIS

The objective of DIS is to develop standards that provide guidelines for interoperability in defense simulations. The Protocol Data Unit (PDU) definition is the first of the DIS standards to be adopted by Institute of Electrical and Electronic Engineers

(IEEE) [IEEE93]. Other aspects of DIS are being refined and extended to address all interoperability issues of distributed simulation.

The DIS standard has three main purposes. The first is to provide a characterization of distributed interactive simulation. The standards address communications architecture, format and content of PDUs, entity information and interaction, simulation management, performance measures, radio communications, emissions, field instrumentation, security, database formats, fidelity, exercise control and feedback. Continued research further clarifies the requirements for the components of distributed simulation. The second purpose is to provide specifications to be used by government agencies and engineers that build simulation systems. The third purpose is closely tied to the characterization. It is to define the terminology of DIS [IST93a]. Terminology is very important as words and phrases that are commonly used in the computer science or communications community have unique meanings in the DIS context. "Tightly-coupled," for example, means two processes are dependent on a common resource (e.g. a particular memory segment). In the DIS context, the same phrase means two simulated entities which are in close physical proximity [IST93b]. Figure 1 shows how specific DIS standards relate to a distributed system framework of application, protocol, and physical network. For instance, the security standard addresses issues at all three levels of the framework.

Recently, the standards have expanded their scope to dual-use applications. Dual-use is a term describing the application of technology to both military and civilian enterprise. The simulation technology used by the military for training and analysis can also be applied to non-military endeavors such as law enforcement, fire fighting, disaster recovery, manufacturing, and entertainment.

D. RATIONALE

Simulators have been developed using different software and hardware architectures coupled with unique protocols for interaction. The DIS protocols are being developed as an international standard to affect meaningful communication and seamless interaction between heterogeneous simulations [IST93a]. DIS used SIMNET as its functional baseline [Lora92]. The scope of the simulated environment in DIS is considerably larger than SIMNET. The DIS environment simulates events on the surface of the earth, both ocean and land, below the surface of the ocean, and in the atmosphere, including space; based on a round earth model. SIMNET simulated engagements on the surface of a flat earth and in the air. [IST93f]

The standards provide guidance for data formats, data representation, model representation, and communication network services. DIS addresses issues relating to network-based simulations. Issues effecting the performance of individual simulation systems are not included in the DIS standards. [IST93a]

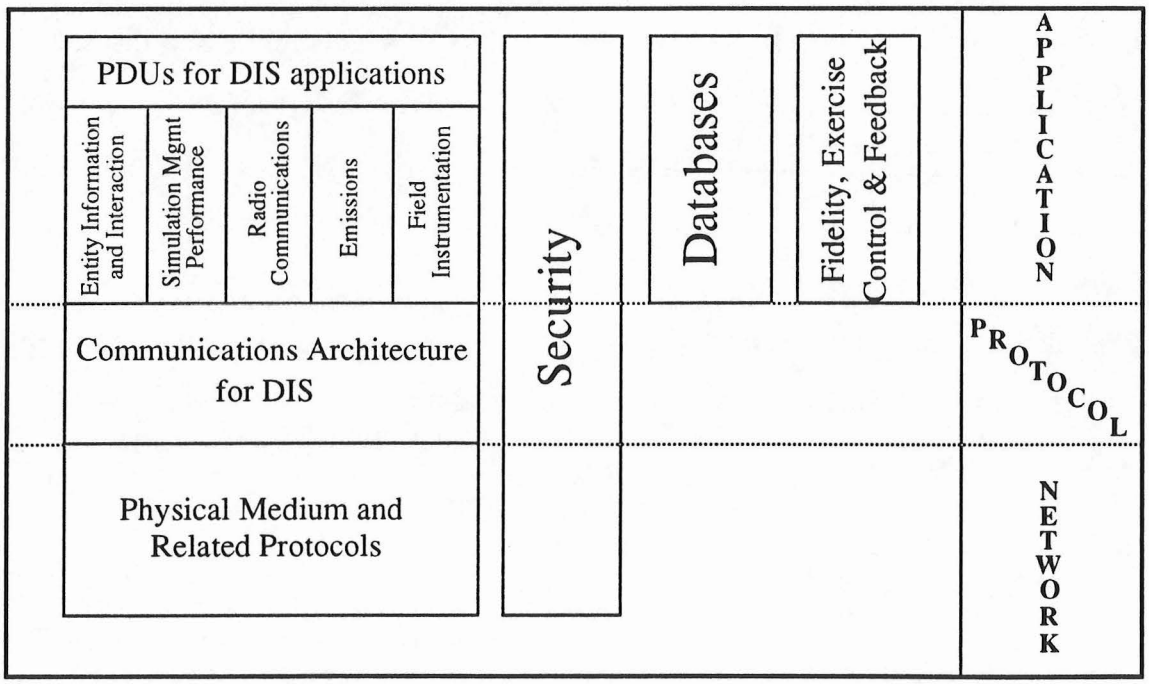

Figure 1 The Standards for Distributed Interactive Simulation (DIS)

E. DIS ARCHITECTURE

In March 1993, the honorable Dr. John J. Hamre of the Senate Armed Services Committee addressed the workshop audience at the Eighth Workshop on Standards for the Interoperability of Defense Simulations. In his comments he compares the architectural magnificence of the Cathedral at Chartre to the architecture being created for DIS. The cathedral is still standing after 900 years. It was designed and constructed using tools and techniques of the day, 1172 A.D., by a host of architects and engineers [Hamr93]. Similarly, the DIS architecture is being designed and constructed with the tools and techniques that are available today. However, the tools and techniques are evolving very rapidly and the expanse of the envisioned simulation continues to grow. To manage the enormity of this task a flexible and extensible paradigm is being adopted. Throughout the literature on DIS architecture, the emphasis is on modular, object-oriented, and reusable designs. [IST93b][Lora92][SRI92]

The DIS architecture is a framework for identifying requirements and specifications for component subsystems and their functional interrelationships. A layered, modular model is used for flexibility, consistency, and extensibility. This model is used to accommodate extensions of the framework and inclusion of new technologies as they become available without redesign.

DIS incorporates the same basic distributed simulation principles as SIMNET [IST93a]. All participating hosts in a networked simulation are responsible for maintaining the state of the entity that they are modeling and the virtual environment. Changes in entity state must be communicated to all other simulation hosts using standard protocols. Each simulation is responsible for the interpretation of messages received from the network. [Lora92]

Figure 2 shows the key elements of the DIS architecture. The functional baseline for warfighting was initially the characteristics and interactions provided by SIMNET. The functional baseline is extended by the inclusion of models that conform to the standards associated with data base formats and functional specifications. Other related issues

complete the architectural description. They include procedures for the verification and validation of DIS compliant simulations, specification of protocols and interfaces for information interchange, and specification of required communication services.

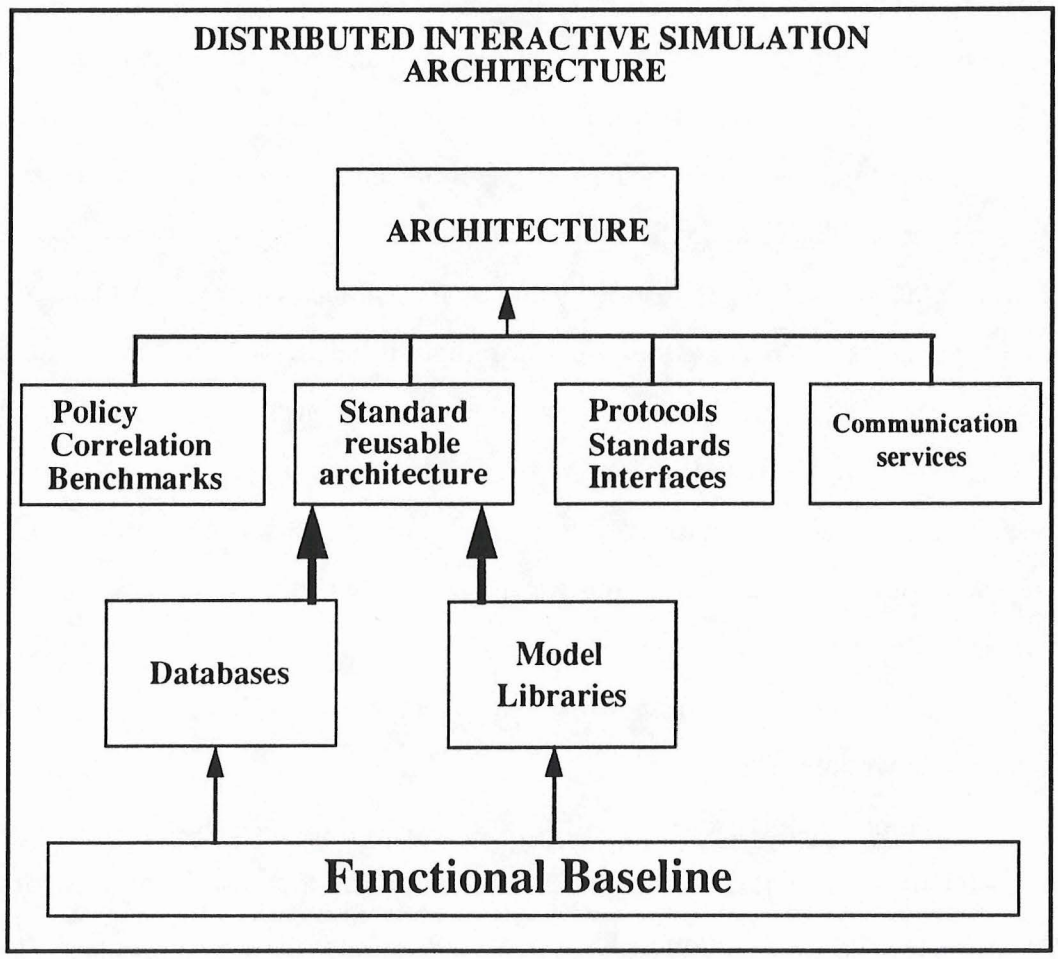

Figure 2 Architecture for Distributed Interactive Simulation

F. COMMUNICATION ARCHITECTURE

A critical component of DIS is the standard for communication architecture. Communication architecture is the description of software and hardware components that comprise a communication system [Stal91]. Like the system architecture, the communication architecture is modular and somewhat open-ended to allow for the

inclusion of emerging technologies (ATM/SONET, light-weight transport protocols) as they mature and become standardized. The standard defines the service requirements for the communication architecture and recommends profiles that provide the services to satisfy the requirements [IST93c]. The standard focuses on using standard protocols, minimizing latency, and providing required levels of service for locally and globally distributed simulation.

1. Assumptions

Basic assumptions about the requirements of DIS simulation and the general structure of DIS are considered throughout this discussion of the communications architecture. The communication architecture must support a variety of devices to include: simulators, stimulators, field instrumentation, large-scale wargame simulators, and all other participants in a DIS. The myriad devices will be operating in multiple simulated exercises dispersed over a large geographic area requiring long-haul communications support. The supporting networks will not be special purpose networks. The same network that supports simulation exercises will support other voice, data, and video traffic that is not related to the simulation exercise. The network will be managed by an agency not affiliated with a simulation exercise or those organizations conducting the exercise. The last assumption is that security is a user issue. It is the responsibility of the user of the DIS simulation to identify risks and countermeasures to protect classified or proprietary data. The standard does not prescribe or preclude the use of specific security mechanisms, however, the standard does prescribe minimum latencies that must be considered when adding security mechanisms to an application.

The approach for developing the communication architecture standard is based on the International Organization for Standards Open Systems Interconnection (ISO/OSI) Reference Model. The reference model provides a framework for modular system design for distributed application protocols [IST93d]. Table 1 shows the correlation of DIS protocols to the layers of the 7-layer model. [IST93b]

Table 1: CORRELATION OF DIS PROTOCOL TO THE ISO/OSI REFERENCE MODEL

Layer Number	Layer Name	DIS Content
7	Application	Type of data exchanged. Rules for determining effects of events (e.g. collision) Remote Entity Approximation (Dead Reckoning)
6	Presentation	Representation of position, orientation, units and encoding
5	Session	Procedure for starting, stopping, joining and exiting an exercise
4	Transport	source and destination process to process addressing packet assembly/disassembly, if required ordering, if required reliability, if required
3	Network	source and destination host addressing
2	Data Link	framing onto physical link logical link control medium access
1	Physical	signals on the medium

The standard employs a phased approach to implementation of communication profiles. It uses proven, widely available, standard protocols in the communication subsystem. We are in phase one of implementation using the Internet protocol suite at the transport and network layers, respectively. Phase two will use OSI/ISO compliant protocols at these layers and phase three will use protocols that comply with the Government Open Systems Interconnection Profile (GOSIP). [IST93d]

2. Communication Service Requirements

Communication service requirements vary based on the need for reliability and the number of destinations. In distributed simulation there are two types of data, each with its own requirements.

The two types of data are simulation and control. In DIS, simulation data is used to communicate entity state updates and coordinate logistics operations. Entity state updates must be sent to all other simulators. This state information must be communicated quickly enough to present a coherent representation of the entity at the destination. Information is broadcast to expedite and disseminate entity state updates.[1] It is sufficient for the entity state information to use a best-effort communication service because an entity's state is continuously transmitted for the duration of the exercise.

Logistic coordination and control data requires reliable, point-to-point communications. Logistics coordination in a simulation is an exchange of requests, offers, and cancellations for logistical support. Control data are messages from the simulation manager(s) to specific "players" to coordinate the execution of the simulation [IST93a]. Best-effort broadcast and reliable point-to-point service requirements are satisfied by using the Internet protocol suite. [IST93d]

3. The Internet Protocol Suite

a. Transmission Control Protocol (TCP)

TCP ensures that communications are completed correctly. The communication services discussed in this section are reliable services because they guarantee delivery of datagrams. TCP provides a point-to-point connection-oriented service. It remembers what has been sent and retransmits the data if it is not received correctly at the destination. If a datagram is too large to fit in one datagram it is segmented

---

1. The standard specifies multicast service based on the assumption that multiple exercises will use the same network. Refer to the discussion of IP addressing in Chapter VI. As a practical matter broadcast, a special case of multicast, is used for our implementation.

into a sequence of smaller datagrams. At the destination, datagrams are ordered, reassembled and delivered to the application process. The destination must send acknowledgments of datagrams that are received intact [Hedr87]. These communication services add to the complexity and processing time of the protocol and are only used when required. Simulation management PDUs use TCP because the simulation manager must ensure that its control information is received by the destination.

b. User Datagram Protocol (UDP)

UDP provides a more streamlined connectionless alternative for transport services and also facilitates multicast [Hedr87]. It is a best-effort communication service. If a datagram is somehow corrupted or lost during transmission, there is no attempt to retransmit the data. The reliable services provided by TCP are not required for entity interaction in DIS. Exceeding dead reckoning thresholds and update timers ensure a continuous flow of entity state update information [IST93a].

c. Internet Protocol (IP)

IP encapsulates the TCP or UDP packet into an IP datagram. UDP/TCP tells IP the IP address of the computer at the other end. The IP header contains source and destination IP addresses, a protocol number to indicate UDP or TCP, and a check sum to confirm uncorrupted transmission.[Hedr87]

4. Performance

Performance standards for communications subsystems are imposed to maintain the coherence of the simulation. As mentioned above, man-in-the-loop simulators function in wall-clock time (real-time) and are measured in the human perception timeframe, which is approximately 100 milliseconds. Update information must be received by all other participating simulation hosts in sufficient time for reading the information from the network, updating the database, and rendering the display. Thus the network can add no more than 100 milliseconds latency for the coherence of the simulation to be maintained [IST93d]. The standard permits an exception for the case when entities are not in close

proximity of one another. In such a case the network can impose up to 300 milliseconds latency. This latency includes the accumulation of delays as the transmission passes through routers, gateways, switches, interfaces, and all other communications equipment. The standard further specifies timeframes for ascending and descending the OSI Reference Model from application to the physical layers and vice versa. Figure 3 illustrates DIS maximum latency in the context of the ISO/OSI Reference Model.

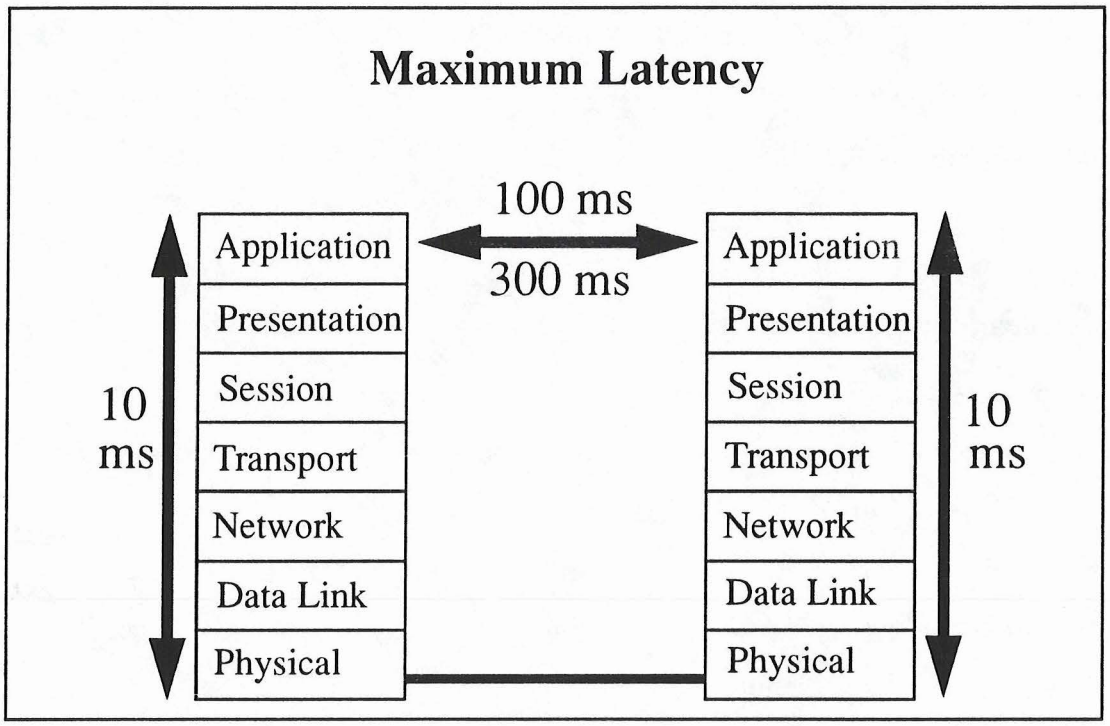

Figure 3 DIS Maximum Latency Specification

G. INFORMATION INTERCHANGE

The DIS Standard for Information Technology, Protocols for Distributed Interactive Simulation Applications, [IST93f], defines the content and protocol for exchanging data messages between simulation applications. The data messages, or protocol data units (PDUs), constitute the user data in the network environment. PDUs convey data pertaining to the state of simulated entities, interaction between entities, simulation management, and

the environment [IEEE93][IST93f]. There are a total of twenty-seven PDUs. Four PDUs are for entity interaction, six PDUs for logistical support modeling, twelve PDUs simulation management, and five PDUs for intelligence and electronic warfare (IEW) modeling. The standards drafting committee has not agreed upon the protocols for simulation management and environmental effects so they are not included in this discussion. [IST93f]

1. Entity State

Host computers require current and accurate information concerning all entities in the simulation. This requirement is satisfied by broadcasting Entity State PDUs containing updated state information [IST93f]. The format of the PDU is tailored to convey all of the information that is required to display an accurate representation of the entity. DIS simulators are required to transmit Entity State PDUs whenever their behavior exceeds the approximated behavior calculated in a predetermined dead reckoning algorithm, or they have not transmitted a PDU in a specified period of time.[2] Entity state PDUs are broadcast to ensure timely dissemination of current information to all other participating simulators.

Entity state information consists of the type of entity, the location in the simulated world, the orientation of the entity, and selected appearance information. Allowable entity types are defined in an associated document, Enumeration and Bit Encoded Values for Use with Protocols for Distributed Interactive Simulation Applications [IST93e]. Entity type is conveyed using a 64-bit record that includes the kind of entity, the domain in which the entity operates, the country that designed the entity, the main category, subcategory, and specific information about the entity. Entity types are constructed in a hierarchical fashion. As an example, the U.S.S. Valley Forge, CG 50 would have an associated entity type record with the information contained in Table 2.

---

2. The maximum time between Entity State PDU transmissions is exercise dependent. In the absence of a specified time, the default is five seconds. [IST93f][Pope89]

Table 2: ENTITY TYPE DATA RECORD FOR U.S.S. VALLEY FORGE

Field	Description	Value
Kind	Platform	1
Domain	Surface	3
Country	U.S.A.	225
Category	Guided Missile Cruiser	3
Subcategory	Ticonderoga Class	1
Specific	CG 50 U.S.S. Valley Forge	4

Each entity is represented by a model based on a set of vertices and their correlation to a center point, or origin, of the model. The location of the entity is the location of the model's origin in the world coordinate system. The DIS standard specifies a geocentric coordinate system with the origin at the center of the earth [IST93f]. The entity orientation is a combination of three angles which represent the yaw, pitch, and roll of the entities local coordinate system in relation to the world coordinate system. These three angles are expressed in radians. Precise communication of this information is necessary to maintain the fidelity of distributed simulation.

Appearance data can add to the realism of the simulation. The standard provides for visual details of an entity, such as paint schemes, battle damage, smoke, lighting configurations, and whether hatches and launchers are raised or lowered. [IST93f]

2. Entity Interaction

There are two kinds of interaction in distributed interactive simulation, operator interaction and entity interaction. Operator interaction is the exchange between the operator and the simulation. Entity interaction is interaction between entities that populate the simulated environment. Entities target, fire upon, and damage other entities in a warfighting simulation. Entities also collide with other entities or the terrain model. Entities

must have a mechanism for communicating these events for realistic simulation. Three types of interaction are currently specified: weapons fire, logistics support, and collisions.

Logistics is an important part of realistic warfighting simulations. For this reason, entity models are created to simulate requirements for logistic support. For instance, vehicle entities are modeled to consume fuel and ammunition; and require repairs. Six logistics PDUs are provided to simulate logistical coordination and replenishment.

Weapons fire information is conveyed using a Fire PDU. The Fire PDU identifies the entity that fired the weapon and the specific characteristics of the weapon that was deployed. DIS provides for both tracked and untracked weapon systems. A tracked weapon system creates another entity, the projectile, when it is activated. When the weapon detonates a Detonation PDU is transmitted. The Fire and Detonation PDUs, and the creation of the projectile entity is the responsibility of the simulator which fired the weapon. The target simulator is responsible for damage assessment in the case where he is hit by a weapon. The target entity then sends an Entity State PDU to notify the other simulators that his state has changed. More detailed information is available in the reference, [IST93f].

III. OVERVIEW OF NPSNET

This chapter provides a general description of the NPSNET project and its achievements as a networked simulation. NPSNET is the simulation application used for the development of the network library.

NPSNET is a real-time, three-dimensional visual simulation system, developed at the Computer Science Department of the Naval Postgraduate School (NPS) in Monterey, California [Zyda92]. It provides a platform for exploration and development of interactive 3-dimensional graphic techniques with the goal of providing a fully-interactive and believable virtual environment that can be configured for many diverse applications. The system's purpose is to develop software for virtual world construction, multimedia applications, and distributed simulation; and to make that software widely available to other government agencies and industry in order to accelerate the development of virtual world technology.

NPSNET is designed as a low-cost simulation system using commercially available, off-the-shelf hardware. Silicon Graphics, Inc. (SGI) IRIS workstations are used to develop and use NPSNET. Networking is accomplished using an SGI implementation of Ethernet for the local area and subscription to the Defense Simulation Internet (DSI) for the wide area.

A. NPSNET IV

NPSNET was completely redesigned during the course of this research. The result, NPSNET IV, was written in C++ using SGI's Performer, an application program interface (API) for graphics applications. NPSNET IV is a real-time, interactive vehicle simulation system in which the user can configure the simulator as an air, ground, nautical (surface or submersible) or virtual vehicle. A virtual vehicle is a non-invasive vehicle that maneuvers in the simulated world but is not represented by a model. We refer to this type of vehicle as a stealth vehicle. The user controls the vehicle by selecting one of three interface devices

which include a flight control system (throttle and stick), a six degree of freedom SpaceBall, and a keyboard. The system models vehicle movement on the surface of the earth (land or sea), below the surface of the sea, and in the atmosphere. Other vehicles in the simulation are controlled by users on other workstations. These users can either be human participants, rule-based autonomous entities, or entities with scripted behavior.

The system is automatically configured for network operations at start-up. This mode presents the greatest challenge to the operator since it matches the operator's skill at fire and maneuver to that of other human participants. The virtual world is populated with up to five different players at NPS (limited by the number of machines in the laboratory), and other participants from around the country. Participation of other sites requires prior coordination for reserving bandwidth on the DSI. NPSNET IV was the simulation system that we used to observe the performance of networked man-in-loop simulations, rule-based autonomous vehicle simulations, and scripted simulations interacting in the same virtual environment.

B. NETWORKING IN NPSNET

Initial networking of NPSNET was achieved using a locally designed network scheme. The scheme used Ethernet and used the basic concepts of SIMNET and DIS discussed in Chapter II. Packet formats were locally designed to transmit information. They were in ASCII format so packet lengths were disproportionately long for the amount of information they contained. This did not present a problem for a distributed simulation on a very small scale. This scheme was used in NPSNET Versions 1 and 2 [Prat93]. The protocol did not require privileged access, but it did not comply with any standard, therefore it restricted use of NPSNET to the local LAN segment.

Another developmental effort of NPSNET was the NPSStealth. NPSStealth is a version of NPSNET that integrated a translator for the SIMNET protocol for interaction over local and long-haul networks. The inclusion of the SIMNET protocol enabled NPSSTEALTH to participate in distributed simulations with other simulations developed

elsewhere using the SIMNET protocol. This implementation was used under strict supervision because the SIMNET protocol required the simulation to run with root privileges. Root access is required because SIMNET requires the use of the 48-bit Ethernet interface address by the application layer. Figure 4 shows the evolution of NPSNET networking.

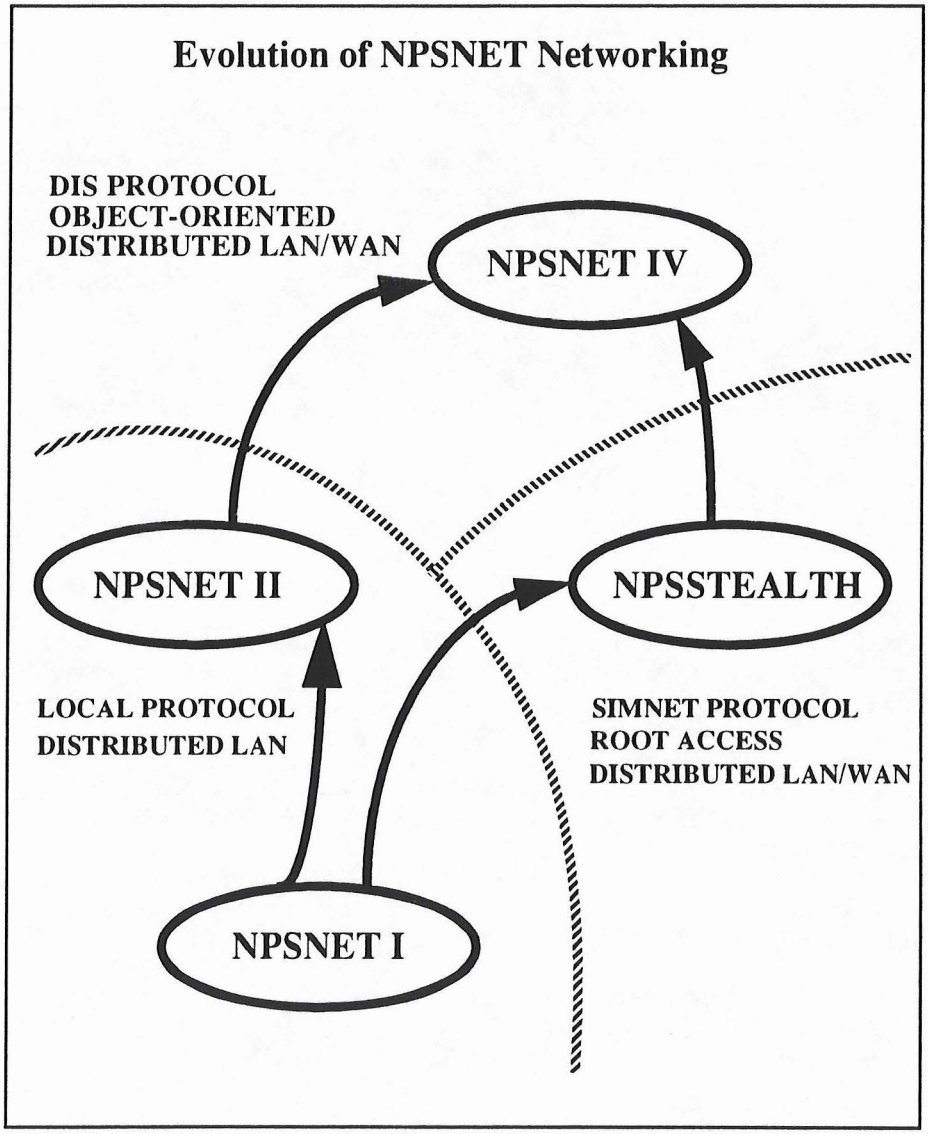

Figure 4 Evolution of NPSNET Networking.

IV. CONSTRUCT OF THE NETWORK ARCHITECTURE

This chapter discusses the construction of a network architecture that satisfied two basic criteria. The architecture must support real-time graphics and it must be DIS compliant.

A. NETWORK ENVIRONMENT

This research was conducted using three network topologies. Each configuration was progressively more complex.

Initially, a small-scale simulation with up to five simulation hosts was conducted in the NPS Graphics and Video Laboratory. This provided a platform for the development and testing of the DIS network library. The laboratory is equipped with three multiport transceiver units (MTU) and SGI workstations with integral Ethernet controllers. The machines are a mixture of multiprocessor and single processor machines.[1] All components are connected by standard transceiver cables. Figure 5 shows the configuration of the local segment.

The second configuration used the T-1 based DSI to link NPS and AFIT, Wright Patterson Air Force Base, Ohio. A BBN T-20 gateway provides access to the DSI. The gateway has an Ethernet controller to bridge communications between the local Ethernet segment and DSI. (Figure 6)

The third configuration was the most expansive. Five sites were interconnected using the DSI and other leased T-1 facilities. In addition to NPS and AFIT, ARPA's Simulation Center and the exhibition booth at the ACM SIGGRAPH conference in Anaheim, California, were connected. NPS, AFIT, and the Naval Research and Development (NRaD) facility in San Diego, California were connected by DSI. ARPA was connected to NRaD

1. The SGI workstations are equipped with different Ethernet controllers. The difference does not effect the interface with the device driver for networking routines. [SGI91]

using their own facilities. NRaD bridged the two networks to a leased T-1 link that terminated in the Anaheim Convention Center. (Figure 7)

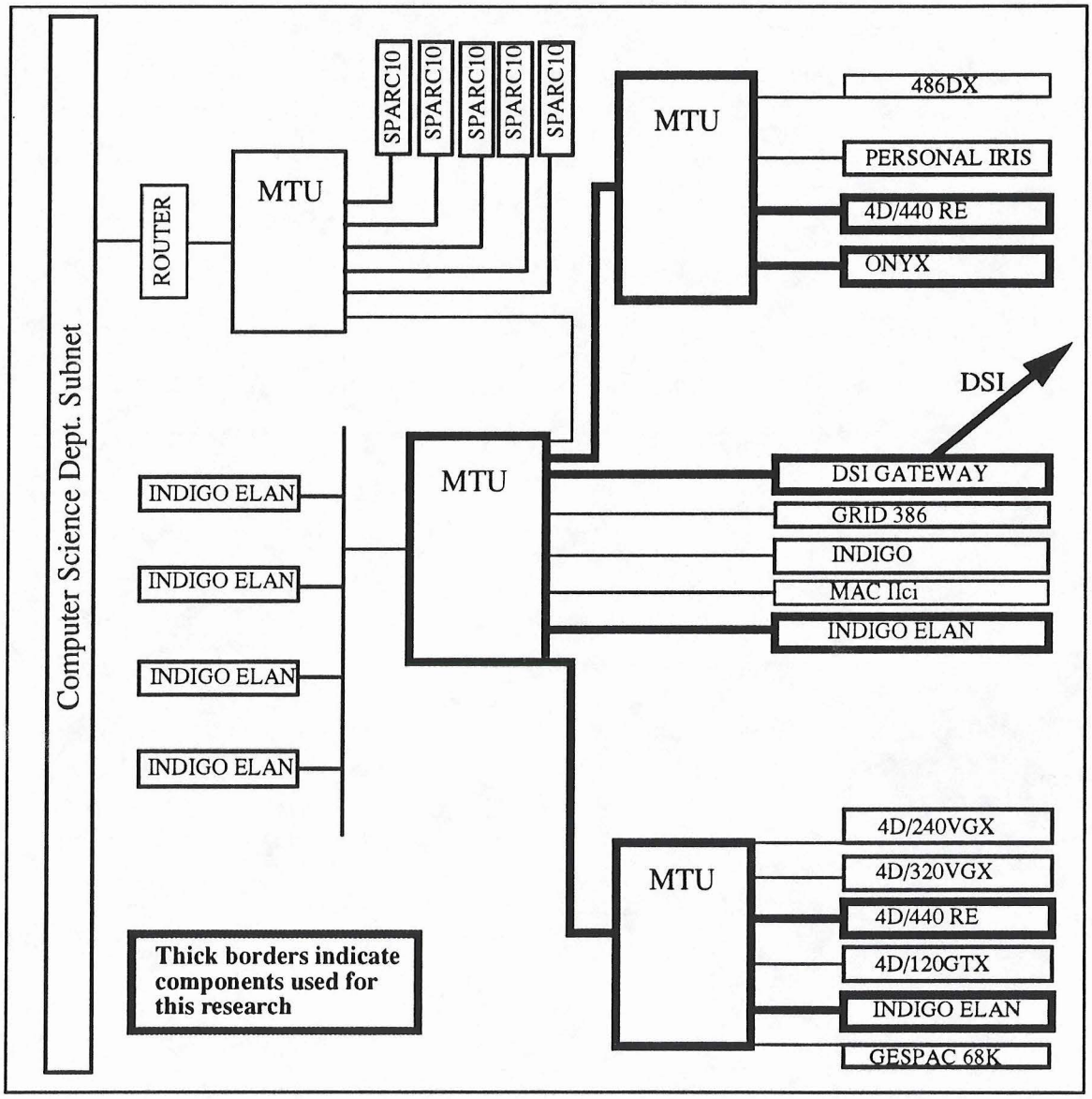

Figure 5 Graphics and Video Laboratory LAN Segment

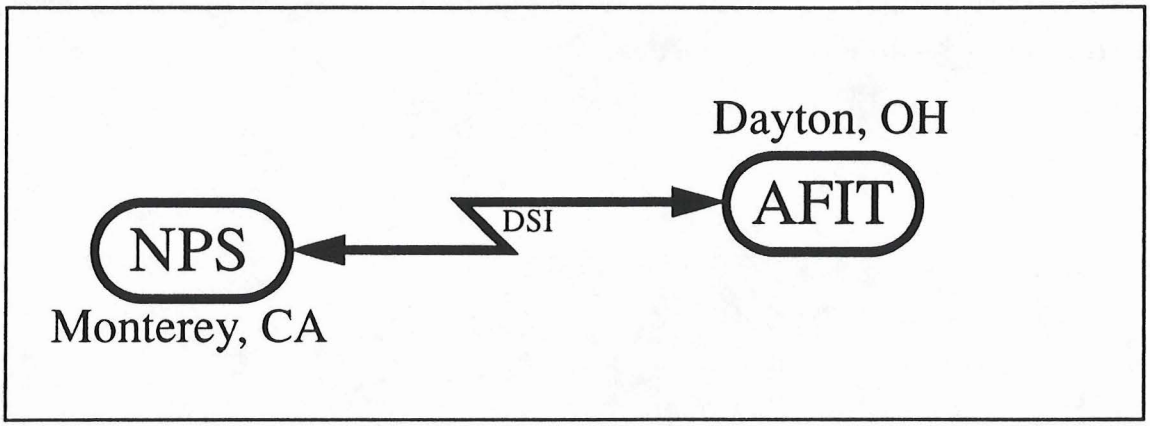

Figure 6 Wide Area Network Configuration 1

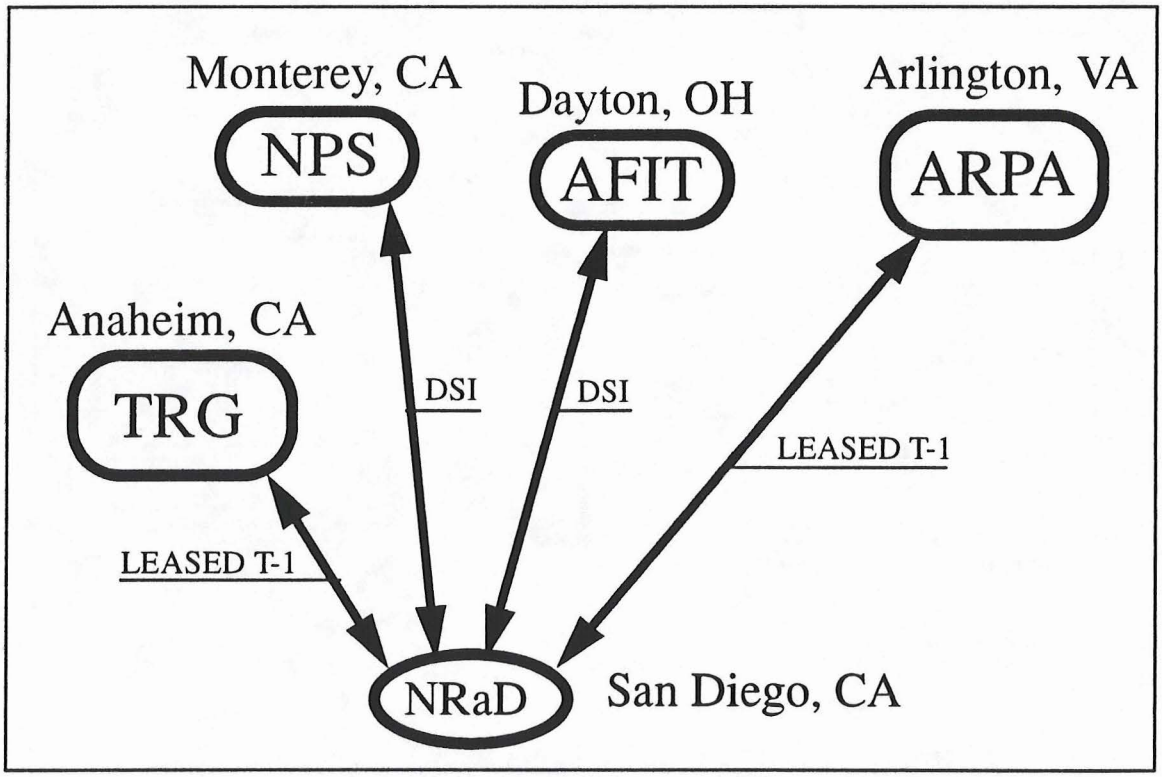

Figure 7 Wide Area Network Configuration 2

B. SOFTWARE ARCHITECTURE

The software architecture has two basic components. The DIS network library provides which provides the interface to the kernel's networking functions; and, a module integrated in the application which contains a set of routines to map DIS PDUs to NPSNET data structures.

1. DIS Network Library

The DIS network library is an extensive redesign of the internal components of John Locke's DIS 1.0 client library [Lock]. The library can be linked to ANSI C or C++ programs. We have maintained the basic user interface of previous implementations of the NPSNET network API: ***net_open(), net_read(), net_write(), and net_close()*** [Lock92]. We have also added a function to allow the user to configure a filter when opening the network, ***net_open_select().*** The functions have been modified to work with our redesigned network harness.

2. Network Harness

A key component of the library is the network harness. When designing the harness, our intent was to minimize the processing overhead and latency incurred in a networked environment by using the multiprocessing capabilities of the SGI machines. The harness resides between the application and the operating system routines that are used for low level network operations. It uses the 4.3 BSD socket-based interprocess communication (IPC) facilities available in the IRIX operating system. [SGI91] We used two sockets, one to receive and one to send DIS PDUs. The sockets are configured for datagrams in the internet domain using UDP/IP. Figure 8 illustrates the conceptual framework of the network harness.

The harness has two functional parts, Figure 8. The first is for sending data. It is invoked by a call to ***net_write()***. This function interacts directly with the kernel. The second part is for receiving data and is implemented as a client-server model using a shared memory buffer for incoming PDUs.

The buffer is located in an *arena*.[2] The data structure in the *arena* is a linked list implemented as a first-in-first-out (FIFO) queue. The server process, ***receiveprocess()***, retrieves data from the network, packages it in a local DIS PDU structure, and writes the PDU to the buffer. This process executes a loop that reads data from the receiving port. The system call *recvfrom()* fits nicely into our scheme because it is a blocking read operation [SGI91]. If the function is called when there is nothing to be read from the network, it blocks, waiting for data to arrive on the network. This facilitates removing data from the network immediately upon arrival, and, since the process blocks it does not consume CPU time. The client process removes a PDU from shared memory and returns it to the application.

3. Basic User Interface

This section provides details of the basic user interface functions mentioned previously. The functions are contained in the file ***client_lib.c()***.

a. net_open()

net_open() is used to initialize data structures for receiving PDUs, configuring send and receive sockets, starting the ***receiveprocess()***, and validating the Ethernet interface provided by the user. The first step is to configure and initialize the arena with calls to ***usconfig()*** and ***usinit()***, respectively. ***usconfig()*** is called to establish the initial size of the arena, 320Kb. ***usinit()*** initializes the *arena* and creates the file **/usr/tmp/NPSNET.net.arena.X**. X is a unique integer to ensure a unique filename.

The *arena* contains the queue and two control features. A binary semaphore is included to protect the queue from coincident access by the client and server. A *barrier* is established as a rendezvous point to ensure the send or receive processes do not begin execution before both sockets are set up. The first process to complete initialization will wait at the *barrier* until the other process arrives. Both processes arrive at the barrier after their respective socket configuration is successfully completed.

2. An *arena* is an SGI facility for memory shared by two or more processes

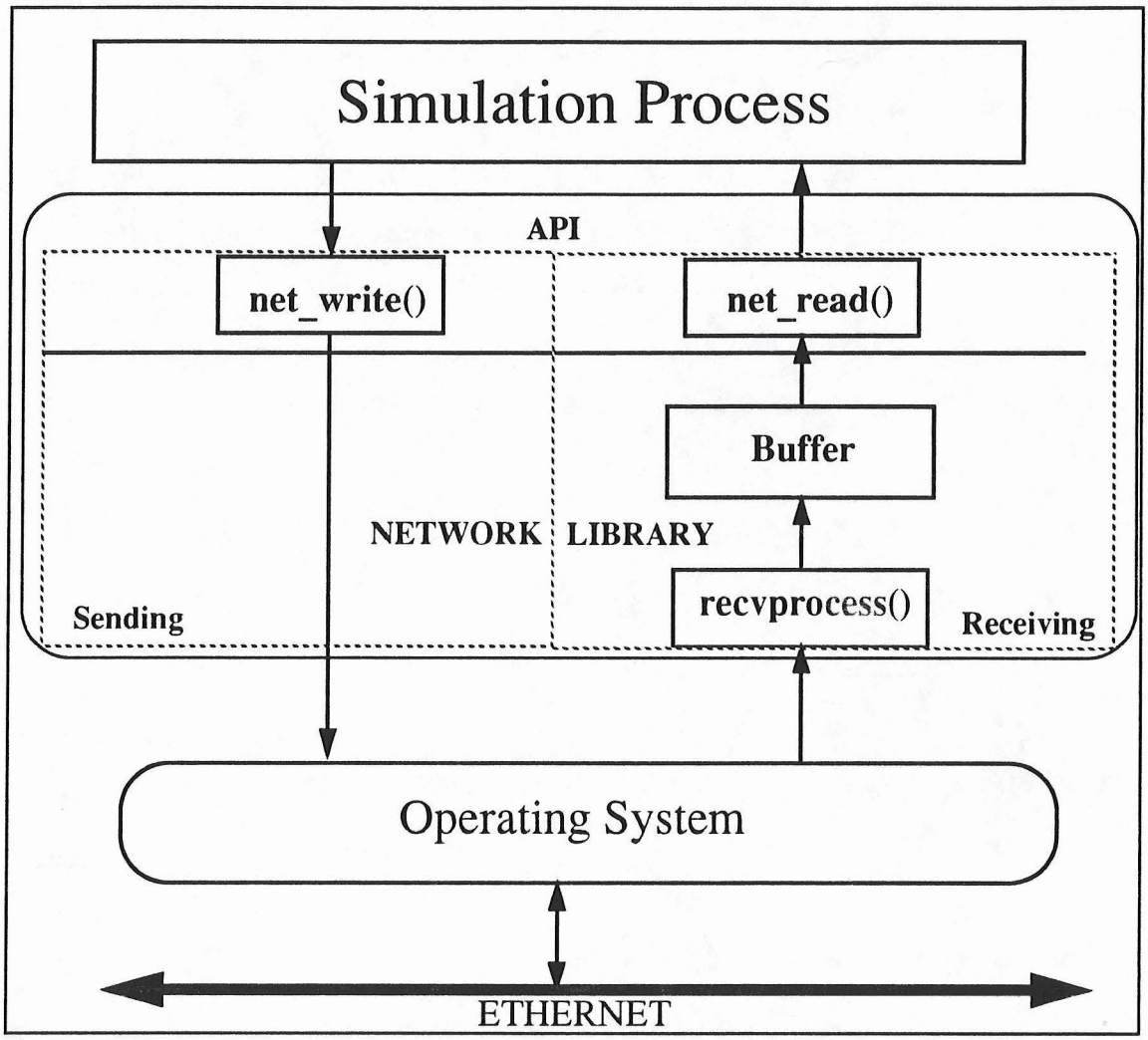

Figure 8 Network Harness

At this point, ***receiveprocess()*** is spawned using ***sproc()***. The process identification is stored in a global variable to be used for termination when the user application terminates. The process opens a socket for receiving data using ***socket()*** and ***bind()***. Once the receive socket is initialized, the process waits for the sending socket to be initialized.

While ***receiveprocess()*** is waiting, the send socket is configured with a call to ***sendsetup()***. ***sendsetup()*** opens a socket for sending data. The send socket is set for broadcasting using ***setsockopt()***. After ***sendsetup()*** returns and providing both sockets were successfully opened, the processes rendezvous at the barrier and proceed.

The last operation of ***net_open()*** is to check the interface controller name argument. ***net_open()*** retrieves the interface configuration structure from the system and compares the system interfaces to the argument. If a match is found, the interface provided in the argument is used. This allows the user to select the interface to be used when the machine has more than one interface.

net_open() returns an integer to the user process. The value one is returned when the network was opened successfully. The value zero is returned when the opening the network failed. The spawned process is terminated when opening the network fails.

b. net_open_select()

net_open_select() is identical to ***net_open()*** with the added feature of selecting PDUs to be read from the network. It also provides for send-only agents.

We want to discard unneeded PDUs as soon as possible. Recall that we rely on the kernel for low level network operations. The earliest that we have access to a PDU is when we read it from the socket. Immediately after reading a PDU from the socket the filtering mechanism is used to discard PDUs.

The filtering mechanism uses a global array and integer. The user provides the quantity and a list of PDU types to be accepted from the network as arguments in the function call. The list is copied into an array that is indexed from zero to the integer provided by the user. This array is used in the ***receiveprocess()*** as discussed earlier.

Send-only agents are supported by an argument of zero in the function call. This means the user does not want to accept any PDUs. This argument is checked before starting ***receiveprocess()***, If the value is zero ***receiveprocess()*** is not started, the send socket is configured and ***net_open_select()*** returns to the calling program.

c. receiveprocess()

receiveprocess() is not a user function, but its functionality is vital to the performance of the network harness. This section discusses the operation of this process.

When both socket are configured, the receive process begins its loop. It first allocates memory to a temporary buffer in anticipation of receiving data, ***malloc(ETHERMTU)***. We use Ethernet maximum transmission unit (MTU) as the size argument for ***malloc()*** to accommodate all known DIS PDUs.[3]

The process then makes a call to ***recvfrom()***, the blocking read. If no packets are present, it blocks. When a packet arrives, the packet is checked to make sure it was not sent by the same host that received it. The packet is then type cast as a PDU header[4] to check the PDU type. If ***net_open_select()*** was used to open the network, the PDU type is compared to the list of acceptable types. If the PDU type is included in the list, then the PDU is processed, if not, the PDU is discarded.

PDU processing is different for variable length and static length PDUs. Table 3 lists the static and variable length PDUs. The variable length PDUs must be unpacked by routines provided in ***recvs.c***. The variable length of a PDU is caused by a variable length array of records in the DIS PDU. The records are fixed length, but the quantity of records vary. As an example, a tank entity may have a number of articulated parts (e.g. a turret, a primary gun). This information is appended to the end of the Entity State PDU. This Entity State PDU has a length of 176 bytes. The base length is 144 bytes and each articulated parameter is 16 bytes. Length = 144 + (2 * 16) [IST93f]. Unpacking is necessary because the network library uses a linked list for this structure.

If the PDU was processed successfully it is appended to the queue in the buffer. This completes one iteration of the loop. The loop continues to execute until a call to ***net_close()*** terminates the ***receiveprocess()***.

---

3. DIS PDU length is restricted to be less than or equal to Ethernet MTU [IST93f]
4. PDU header types are defined in pdu.h

Table 3: VARIABLE AND STATIC LENGTH PDUS

Variable Length	Static Length
Entity State	Fire
Detonation	Resupply Cancel
Service Request	Repair Complete
Resupply Offer	Repair Response
Resupply Received	Collision
Action Request	Create Entity
Action Response	Remove Entity
Data Query	Start/Resume
Set Data	Stop/Freeze
Data	Acknowledge
Event Report	Laser
Message	
Emission	
Transmitter	
Signal	
Receiver	

d. net_read()

net_read() is used to retrieve a PDU from the buffer. This is the client process in the network harness discussion. It acquires the semaphore and takes the PDU[5] from the head of the list, providing the list is not empty. It then releases the semaphore.

5. Note that the network harness allocates memory for incoming PDUs. The network harness deallocates memory for its list structure, but, the memory containing the PDU remains and must be deallocated by the user process to prevent memory leaks.

The function returns an integer value, a handle to a PDU, and a PDU type. The PDU type is read from the PDU header record. The integer value is zero if the list was empty; greater than zero if the list was not empty, and negative one if the operation failed.

e. net_write()

net_write() is used to transmit PDUs. The functions arguments are a handle to a PDU and a PDU type. First the PDU header record information is completed. The header record includes the exercise identification number, the PDU type, a time stamp, and the protocol version.

As discussed in ***receiveprocess()***, variable length PDUs require processing due to the list structure that we use for appended features. Functions are provided in the file ***sends.c*** to remove imbedded structures from variable length PDUs before transmission. These functions complete the header record with the PDU length and pass the PDU to ***send_it()***. Static length PDUs have their header records completed and are passed directly to ***send_it()***. ***send_it()***, in file ***sends.c***, makes the ***sendto()*** call to deliver the PDU to the kernel.

f. net_close()

net_close() is used to terminate network operations. It closes the send and receive sockets using ***close()***, and terminates ***receiveprocess()*** using ***kill()***.

4. Network Utilities

netutil.cc contains the function ***parse_net_pdus()*** which makes the call to the network interface function ***net_read()***. The file also contains the functions that initialize and manage the NPSNET data structures for incoming DIS PDUs. The two primary structures are a vehicle hash table and an entity type tree.

The hash table is used to find an index into the NPSNET entity array. The hash function uses the site, host, and entity fields in the EntityID record to find a location in the table. The location is in one of three states: empty, available, occupied. Empty and occupied are self explanatory. Available indicates that the location is available for use because a

vehicle has been removed from the table, but the next location in the table is occupied. This mechanism reduces the chance of duplicating a vehicle in the table. Occupied locations are checked for their EntityID. The table is searched sequentially (circularly) until a match or an empty location is found. It is assumed that the PDU represents a new vehicle when an empty location is found. When it has been determined that the entity is a new vehicle the entity type must be determined.

The entity type tree is used to map DIS entity types to NPSNET vehicle types when a new entity is introduced to the simulation. The tree is designed to model the hierarchy used for the definition of DIS entity types [IST93e]. The top level of the tree corresponds to the fields in the entity type record [IST93f].It is initialized with data from the dynamic models file. The tree is traversed to find the type of the new vehicle which serves as an index to the NPSNET vehicle types array.

The library also includes three programs that are useful for troubleshooting and monitoring the performance of DIS distributed simulations. ***print.c*** contains print routines for DIS PDUs and the component records of PDUs. These routines are used in the program ***test_it.c*** to decode packets. ***pacrate.c*** is a program that monitors the packet transmission rate of all simulators on the network. It uses five second samples to report the average transmission rate of entities in the simulation. Entities are uniquely identified by their site, host, and EntityID. We used this program to ensure that our dead reckoning algorithms were functioning properly to prevent excessive network loading.

5. Packing and Unpacking PDUs

recvs.c contains the functions to unpack variable length PDUs. Unpacking is required because we employed a linked list structure for the elements of a PDU that causes the size to vary. The functions are called by ***receiveprocess()*** and all have the same basic operations.

The routines allocate memory to a buffer using the functions in ***mallocs.c*** (e.g. ***mallocEntityStatePDU()***). The size of the buffer is determined by the PDU type and

structure[6] which are a fixed length in our implementation. They include a pointer to a list structure for elements that would otherwise cause the length of a PDU to vary. These elements could be articulated parameters for a vehicle or the number of supply types on a resupply offer. The basic, or fixed length, part of the received PDU is copied into the buffer. All variable length PDUs contain a field that tells the number of elements (e.g. num_articulat_params) that are appended to the PDU [IST93f]. The elements are fixed length. Memory is allocated for each element (e.g. ***mallocArticulatParamsNode()***), the element is copied from the incoming PDU, and attached to a list. When the element list is complete, it is attached to the pointer in the buffer (e.g. articulat_params_head). This completes the unpacking and the PDU is returned to ***receiveprocess()***.

sends.c contains functions that pack variable length PDUs from our local structures to DIS PDU structures and the kernel interface for network operations. The packing functions remove the information from our list structures (e.g. articulated parameters) and append it to the end of the bit stream that will be transmitted.

Memory is allocated for the PDU transmission based on the fixed length of the PDU[7] and the number of additional elements. Recall that the elements all have the same structure with a fixed length.

size = base length + (number of elements * size of element)

First the fixed length data elements are copied into a buffer, then, if the number of appended elements is greater than zero, the data from the list structure is appended to the buffer. When complete the pointer to the buffer is passed to the ***send_it()*** function for transmission.

send_it() is the interface to the kernel and underlying network for all transmissions. Its arguments are a pointer to a buffer and a length. The buffer contains the

---

6. PDU structures are defined in pdu.h.
7. In computing the base length we found that ***sizeof()*** returns four extra bytes which caused the data to be corrupted and the PDU to be too long. To resolve this we did not use the sizeof() function. We instead used a defined base length for PDUs with variable elements. These definitions are in pdu.h.

PDU. It transmits the PDU by calling ***sendto()***. The function returns a zero if the operation failed, otherwise it returns a one.

6. Memory Management

The file ***mallocs.c*** contains memory allocation routines that are used when receiving PDUs. The functions are tailored to the PDU types and structures that we have defined in pdu.h. Their construct allocates memory using ***malloc()***. On successful completion they return a pointer to the desired PDU type. When the allocation fails a null pointer is returned.

In addition to the functions for specific PDU types there is a general purpose function, ***mallocPDU()***, that can be used to allocate memory for all PDU structures. The function takes a PDU type as an argument. The argument is used to call the specific memory allocation function.

The implementation of NPSNET IV uses these functions to allocate memory for temporary structures to send PDUs. As an example, there is a sequence of function calls that NPSNET uses to send an EntityStatePDU. First a pointer to an EntityStatePDU is declared and initialized with a call to ***mallocEntityStatePDU()***. This function returns a pointer to an EntityStatePDU structure in memory. Then the structure is loaded with the data to be transmitted and the pointer is passed to ***net_write()*** to transmit the PDU. After ***net_write()*** returns successfully, the memory for the temporary structure must be deallocated using ***freePDU()***. The argument to ***freePDU()*** is a PDU type.

The file ***free.c*** contains the routines to free memory that was allocated for PDU structures. For PDUs with appendages, the appendages are freed first. Then the memory used by the basic structure is released. The discussion in the preceding paragraph provides an example of how ***freePDU()*** is used.

C. SUMMARY

This chapter covered the construct of our DIS compliant network architecture that supports real-time graphic simulation in a distributed environment. The architecture was used on three network topologies based on Ethernet and a T-1 backbone wide area network.

The software has two basic components, the DIS network library and an application module, ***netutil.cc***, that efficiently maps DIS data to NPSNET data structures. The DIS network library is built on an optimized harness that uses socket-based IPC and takes advantage of our multiprocessor hosts. The API has five basic functions: ***net_open()***, ***net_open_select()***, ***net_read()***, ***net_write()***, and ***net_close()***. The library includes supporting programs and routines for memory allocation/deallocation and monitoring distributed simulation. The library is portable and can be linked with ANSI C or C++ programs.

V. USING THE DIS NETWORK LIBRARY

The DIS Network Library incorporates the DIS Version 2.0.3 standards for communication architecture and PDU formats. This chapter discusses the use of the library. Each function description begins with a function prototype and includes some sample code.

A. HEADER FILES

The library's header files are located in **~zeswitz/network/h**. These header files contain definitions of the information that is in the Enumeration and Bit Encoded Values document [IST93e]. Two of the header files deserve particular attention. The file ***disdefs.h*** contains all common information used for the network harness. ***disdefs.h*** must be included in the user program when linking to the library in **~zeswitz/network/bin**. The header file also includes all required header files to recompile the library. ***pdu.h*** contains the data structures that we used for conveying PDUs between the application and the network harness.

B. USING net_open()

prototype: int net_open(char *interf);

The function net_open() is used to initialize the network for a user program. The user must specify the name of the interface to be used.[1] It returns a value of one if the network opened correctly and a value of zero if the network opening failed. An example of the function call is:

```
char interf[3] = "et0";
/* initialization routine */
if (net_open(interf) == FALSE)
   printf("net_open failed");
```

1. If the interface name is not known, it can be obtained by executing the user command **netstat -i**. The command will display the system's interface names.

A user can not have more than one process use a port in our configuration[2]. If you attempt to bind a socket the second time, the system call, ***bind()***, will fail and a message, "**bind(sock_recv): port busy**" is displayed to the standard output device. The user must identify the process that is using the socket and preempt that process or assign a different port and recompile.

C. USING net_open_select()

prototype: int net_open_select(char *,int, short *);

The function net_open_select() allows you to specify the types of PDUs to be processed by the network library routines. The user application passes the interface name, the quantity of PDUs to be received, and an array of PDUs as arguments.

```
char interf[3] = "et0";
int how_many = 3;
short which_PDUs[3];

which_PDUs[0] = EntityStatePDU_Type;
which_PDUs[1] = FirePDU_Type;
which_PDUs[2] = DetonationPDU_Type;

/* initialization routine */
if (net_open_select(interf,how_many,which_ones) == FALSE)
   printf("net_open_select failed");
```

This function can also be used to create a send-only agent by using a quantity argument of zero. The process that reads PDUs from the network is disabled when the network harness is configured to send-only.

D. USING net_read() and freePDU()

prototype: int net_read(char **, PDUType *);

prototype: void freePDU(char *);

---

2. For simulators this is acceptable since there is likely to be only one simulation running on a simulation host.

The function ***net_read()*** is used to retrieve a PDU from the network process. It returns a pointer to a PDU, the PDU type, and an integer value. The integer value is greater than zero if the operation is successful.[3] A value of zero indicates there were no PDUs in the queue to be retrieved. A value of negative one indicates the function call failed to read the queue correctly. After the PDU has been processed by the user program, the memory allocated to the PDU structure must be freed to prevent memory leakage. The function to free a PDU is ***freePDU()*** in the file ***free.c***.

```
int nodes;
char *pdu;
PDUType pdu_type;

while (0 < (nodes = net_read(&pdu, &pdu_type))) {

    /*  successfully read PDU from the queue */
    /* the user program must deallocate the memory used by the pdu
       after the pdu has been processed                              */
    .
    /* process PDU */
    .
    freePDU(pdu);
}

if (nodes == -1) {
   printf(" error in net_read()\n");
```

E. USING net_write() and mallocs.c FUNCTIONS

prototype: int net_write(char *, PDUType);

prototype: char *mallocPDU(PDUType); or

char *malloc<PDUType>();

---

3. The positive integer returned from ***net_read()*** indicates the number of PDUs that were in the queue before the function was called.

The function ***net_write()*** is used to send a PDU to the network process. The discussion of the ***mallocs.c*** routines are included because they allocate memory for the temporary structure that is loaded then transmitted.

Prior to sending a PDU to the network, a function in ***mallocs.c*** is called to allocate memory to a temporary structure that is loaded with user data, then transmitted using ***net_write()***. The arguments to ***net_write()*** are a PDU pointer and a PDUType. The function returns the value one if it is successful and the value zero if the operation fails.

```
EntityStatePDU *ESpdu;

ESpdu=(EntityStatePDU *) mallocPDU (EntityStatePDU_Type);
.
/* fill in the fields */
.

if (net_write(ESpdu, EntityStatePDU_Type) == FALSE)
    printf("error in net_write()\n");

free(ESpdu);
```

F. USING net_close()

prototype: void net_close();

net_close() terminates network processes. No arguments are required.

VI. EXPERIMENTAL RESULTS

Our experiments focused on: (1) determining whether the semantics and syntax of the protocol was properly implemented; and (2) informally gauging the performance of the network harness and DIS implementation relative to the number of hosts and entities participating in a given simulation. From these experiments, we concluded that our software functioned properly and we extrapolated the upper bounds of network performance given our current hardware and software configuration (Chapter IV). The primary measure of network performance was the observed performance of the simulators.

We employed a variety of network monitoring tools for our experiments. SGI NetVisualizer provides facilities for recording packet counts, byte counts, and Ethernet capacity measurements. TCPDUMP, a public domain network monitor developed at University of California, Berkeley [Jaco92], performed initial connectivity tests. Both monitors gather data at the hardware interface level. We developed other programs to monitor the network from the application level. These tools decode and generate packets, and measure packet rates. We conducted experiments in three phases (as previously discussed in Chapter IV). During the first two phases, our simulations contended for network resources with the other hosts on the network. No effort was made to isolate simulation hosts from other network traffic. In the third phase modifications were made to the Ethernet to increase the probability of a successful demonstration.

A. PHASE I: NPS LOCAL SEGMENT EXPERIMENT

The first phase was conducted on the NPS Graphics Laboratory Ethernet segment using from one to four simulation hosts. The experiments were primarily connectivity tests to ensure DIS compliance of the network library routines. We regularly observed Ethernet frames containing DIS PDUs encapsulated in UDP/IP packets on the network. The content of the DIS packets was consistent with the data loaded by the application in accordance with DIS protocols. However, the length was incorrect due to the linked list structure used

for variable length PDUs (as discussed in Chapter V). We corrected this deficiency and observed from continued experimentation that the DIS network library was working properly.

We experienced our worst performance during local segment tests. There were four participating hosts and a sound server on the network. The queue of received PDUs grew to over 600. Graphics displays halted and the sound server became overloaded. The reason appeared to be that the simulation was not sufficiently insulated from other activities supported by the laboratory LAN segment. In addition to the simulation, three other hosts were executing an interactive animation program that resides on a simulation host. Four other hosts were being used for artificial intelligence research. All hosts on the segment use the file server, an Indigo Elan, which was also the sound server. The harness continued to queue incoming packets in spite of the heavily loaded network. Since the queue length grew to 600, it appeared that the latency was not caused by the network harness and its associated processes, but instead, by host process scheduling. After this experience we minimized incidental use of simulation hosts during testing and the problem did not reoccur.

B. PHASE II: NPS AND AFIT USING DSI

The second phase of testing was conducted with AFIT during five separate experiments over a period of one month using the DSI. DSI linked two LAN segments together revealing the importance of IP addressing. Because we were broadcasting, the destination IP address was also a broadcast address. Broadcast addresses are unique to a local network and can not be legally duplicated on other subnetworks. For multiple remote LANs to communicate in broadcast mode they must be using the same broadcast address. Specifically, hosts on the Graphics Laboratory Ethernet segment use an IP broadcast address of 131.120.7.255. Hosts on the AFIT Ethernet segment broadcast to 120.17.56.255. When a packet is broadcast over DSI from NPS to AFIT, the packet will have an IP destination address of 131.120.7.255. As this packet ascends to the IP layer on an AFIT host, the address is not recognized and the packet discarded. To resolve this problem we

configured the Ethernet hardware interface to broadcast to a common IP address and added a routing table entry to direct the specially addressed packets to the local host. This technique was applied to each participating host. Having satisfied the technical requirements of the UDP/IP protocols we were able to reliably exchange packets containing DIS PDUs with AFIT.

During the second experiment the network harness failed on the receipt of zero length DIS PDUs causing a segmentation fault. Though the PDUs were the result of an error in the application transmitting them, we added code to handle the condition gracefully. We initialized the incoming PDU buffer to all zeros before reading a PDU from the socket.

This second phase was in preparation for the ACM SIGGRAPH demonstration. Load testing was minimal because the experiments involved one or two simulations at each site to expedite the resolution of simulation fidelity issues.

C. PHASE III: ACM SIGGRAPH 93

The third phase was the ACM SIGGRAPH demonstration. The general configuration was described in Chapter IV. The T-1 was multiplexed with a televideo conference restricting simulation bandwidth to 704 Kilobits per second.

Our intent was to reduce the amount of non-simulation traffic. During this phase we distributed all required files to the participating hosts to eliminate file server access. We also took a different approach to IP addressing. Each host was configured with the IP address from an ARPA LAN segment. By using the ARPA addressing our network received ARPA's non-simulation traffic. Simulation traffic accounted for 65% of the total load when we were connected to ARPA. When the T-1 would lose synchronization, simulation traffic accounted for 100% of the total load.

D. LOAD ANALYSIS

Our primary test of network performance was visual observation of the smoothness of the simulation displays, which can be degraded by network latency. In addition, an informal load analysis was conducted to approximate an upper bound of the number of

simulation hosts and entities that can reliably participate on our Ethernet segment. We use *offered load* as our performance parameter, defining it as the percentage of total bandwidth that a simulation host used at peak packet transmission rates. Packet transmission rates and, thus, offered load depends on the number of hosts, the number and type of entities, the scenario, the dead reckoning algorithm and thresholds, and the activities in which entities are engaged [IST93c][Prat93].

Our simulation hosts were used for man-in-the-loop and semi-autonomous simulations. Man-in-the-loop simulation hosts model a single high-performance aircraft and its weapon systems (gravity bombs, missiles, and guns). The aircraft models were not physically-based and, thus, were capable of performing some incredible, if not impossible, flight behavior (e.g. stopping in mid-air). All weapon systems were modeled as guided munitions. At any given time, a single host was limited by the application to generating PDUs for up to six entities (1 aircraft and 5 weapons). The semi-autonomous simulation hosts model multiple air, ground, or surface vehicles.

1. Packet Rates

Graphic simulations are capable of generating Entity State PDUs at a peak rate of one per frame. For example, a simulation that displays thirty frames per second can generate thirty Entity State PDUs per second. Weapons were restricted to one firing per second. Upon a firing, the simulation generates two additional packets: one Fire PDU to establish the firing and one Entity State PDU to model the munition. Upon munition impact one Detonation PDU is transmitted. Potentially, a simulation can generate forty-one PDUs per second; thirty Entity State PDUs for the aircraft, five Entity state PDUs for munitions, one Fire PDU and five Detonation PDUs. However, during experimentation we observed a peak rate of twelve Entity State PDUs per second for the aircraft. The packet rate was constrained by dead reckoning thresholds. In addition to the twelve packets per second, the aircraft simulator generated at most three packets for a weapon firing sequence totalling

fifteen packets per second. The peak packet transmission rate for this type of simulator is fifteen packets per second.

Semi-autonomous simulation hosts modeled multiple slow moving vehicles. One simulation modeled three sailboats on a lake; another modeled twelve aircraft slowly circling at different points in the world; a third modeled twelve ground vehicles moving in formation. These models generate packets at a peak rate of five packets per entity per second.

2. Packet Length

Packet lengths were small relative to the 1500 bytes that Ethernet allows for user data. We transmit one Entity State, Fire, or Detonation PDU per Ethernet frame. The Entity State and the Detonation PDUs are variable length by definition, but had a constant length in our application since we did not use the variable length features. Implementing UDP/IP/Ethernet adds forty-six octets of overhead to each frame of data transmitted.[1] Table 4 shows the base length of the DIS PDU types, the overhead associated with this network architecture, and the total frame length.

Table 4: TOTAL FRAME LENGTH (IN BYTES) OF DIS/UDP/IP/ETHERNET FRAMES

DIS PDU Type	Base Length	UDP	IP	Ethernet	Total Length	% of Ethernet MTU
Entity State	144	8	20	18	190	12
Fire	88	8	20	18	134	9
Detonation	104	8	20	18	150	10

3. Simulation Bandwidth Utilization

The bandwidth required for the three phases of distributed simulation experiments was predictable based on the previous discussion. In the third phase there was a maximum

1. A frame is a packet that has been transmitted on the communications medium.

of eleven hosts simulating a maximum of 50 entities during the free-play scenario. Seven of the simulators were high-performance aircraft. Four of the simulators modeled multiple slow moving vehicles. The average packet length was 190 octets, including network overhead. Simulation traffic peaked at 168 packets per second accounting for 2.5% of Ethernet and 16% of T-1 bandwidth. Total utilization peaked at 3.2% of Ethernet and 20% of T-1 capacity. This allows us to approximate the number of entities that can be simulated on our current network configuration.

We approximate a maximum of 1093 entities could be simulated using Ethernet assuming the same ratio of slow to fast moving vehicles and that optimal Ethernet performance can be achieved (Figure 9). A more reasonable expectation is simulation of 312 entities with a network load of 20%. Using a T-1 (1.544 Mbps) long haul service restricts Ethernet use to approximately 15% for 234 entities. NPSNET IV, which currently allows 200 entities, will use no more than 12.8% of Ethernet and 83% of T-1 bandwidth. [Stal91]

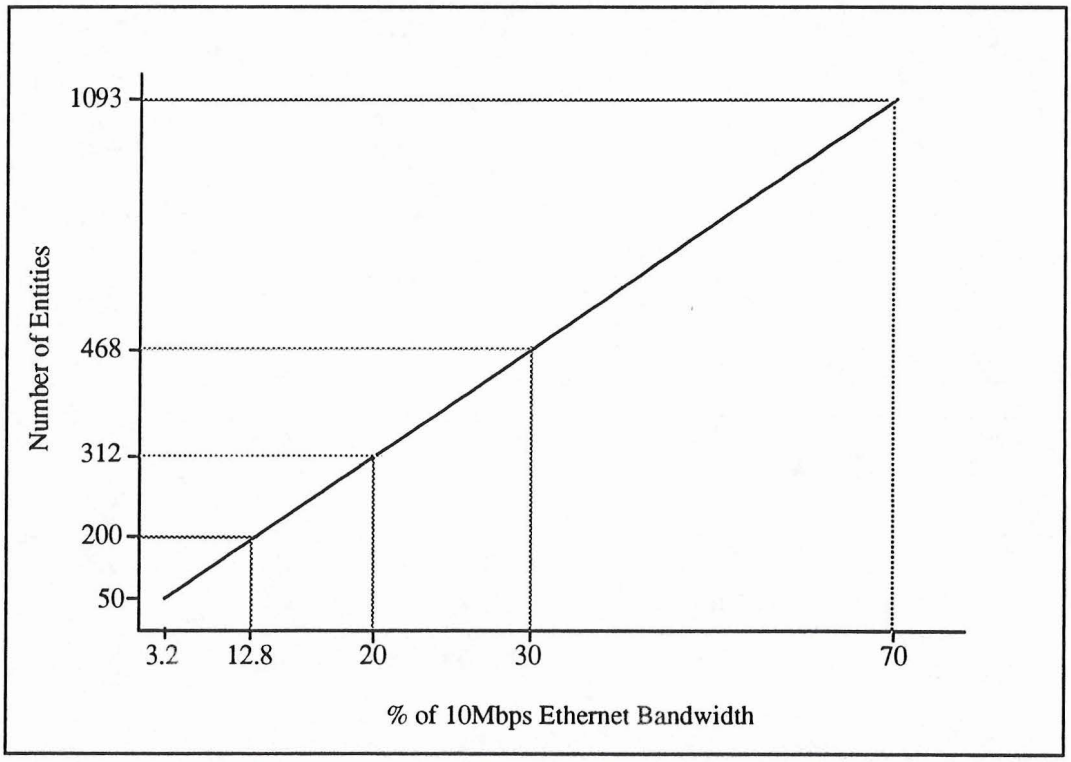

Figure 9 Ethernet Bandwidth Approximation Based on Number of Entities

These approximations establish the upper bound of the number of entities that can be simulated using our current Ethernet/T-1 configuration. Many other factors contribute to effective throughput and need further study.

VII. CONCLUSION AND TOPICS FOR FUTURE RESEARCH

A. CONCLUSION

The objective of this research was to develop a library of network routines that: (1) comply with the DIS communication architecture and information interchange standards; and (2) support distributed real-time simulation. The library was developed and successfully tested in three network configurations. We have reached the following conclusions:

- The library routines comply with the aforementioned standards
- Ethernet is capable of supporting distributed real-time simulation on a small to moderate scale
- The first network bottleneck will be the T-1 wide area network
- Distributed simulation performance can be improved by insulating simulation hosts and network bandwidth from non-simulation traffic

B. TOPICS FOR FUTURE RESEARCH

This research provides a baseline for study and development of network configurations for distributed simulation. The following is a list of topics for future research.

- further optimization of the network harness
- implementation of the other DIS standards
- aggregation of DIS PDUs in single network frames
- formal network load studies
- formal studies of packet loss rates
- implementation of networks that provide bandwidth reservation
- implementation of IP multicast

APPENDIX A: DIS NETWORK LIBRARY USER'S GUIDE

DIS NETWORK LIBRARY

Steven R. Zeswitz, John Locke, Mike Macedonia and David R. Pratt

Department of Computer Science
Code CS/Pr
Naval Postgraduate School
Monterey, California 93943
pratt@cs.nps.navy.mil
Fax: (408) 656-2814

Overview

The DIS network library was developed to provide a network interface for DIS simulation systems using SGI workstations communicating through Ethernet. It has been tested in a number of network configurations and should be portable across any SGI workstation. This document is designed to give the user the ability to configure the simulation host and invoke the application program interface (API) routines that interact with the network.

The library includes data structure definitions for DIS Version 2.0.3 PDUs. The library routines manage the conversion between memory data structures and the corresponding network packet formats and provide a number of memory management functions. The code is written in C and supports programs written in C or C++.

Please let us know of any improvements for the library. We keep a record of all known users and provide e-mail announcements of updates to the library.

Capabilities

The library supports asynchronous communication between DIS processes. It is based on a network harness designed for multiprocessor machines using the 4.3 BSD socket-

based interprocess communication facilities of the IRIX operating system. Sockets are configured for UDP/IP broadcast.

The library API consists primarily of five basic routines: ***net_open()***, ***net_open_select()***, ***net_read()***, ***net_write()*** and ***net_close()***. ***net_open()*** and ***net_open_select()*** opens the sending and receiving ports, configures an arena (shared memory), initializes a semaphore and creates the process that reads PDUs from the network. ***net_open_select()*** permits the user to filter unwanted PDUs as soon as they are read from the socket. The arena is used to queue incoming PDUs. ***net_read()*** retrieves a PDU from the head of the queue. ***net_write()*** transforms the PDU structure to network structure, if necessary, and writes the outbound PDU to the socket. ***net_close()*** closes the sockets and terminates the receiving process.

Memory management routines are provided for DIS PDU structures contained in the library. The functions are tailored to specific PDU types.

Directory Structure

The network library directory structure is self-contained in **~zeswitz/network**, with the exception of the standard header files from /usr/include. The Makefile uses relative path addressing.

The header files are located in **~zeswitz/network/h**. The file ***disdefs.h*** contains all common definitions used in the library routines including the UDP send and receive port numbers, the exercise identifier, and the default Ethernet interface name. pdu.h defines the memory structures for PDUs. The other header files contain the defined constants for DIS.

The source code is located in **~zeswitz/network/src**. The API functions are contained in the file ***client_lib.c***. The supporting memory management, reading and writing functions are contained in the other files in this directory.

The executable library is located in **~zeswitz/network/bin**. Programs can invoke the network routines by linking to this library.

Some network monitoring utilities are located in **~zeswitz/network/utils**. These programs monitor packet rates, decode packets, and log all packets from an exercise.

Using the Network Library

The network library is simple to use. ***disdefs.h*** must be tailored to the user application. The user must specify the send and receive port addresses for UDP if the defaults of 2999 and 3000 are not satisfactory. If ***disdefs.h*** is changed, the library must be recompiled. This header file must be included in the application program and the program must link to the library.

Performance and Limitations

There are no known limitations at this time.

Acknowledgments

We wish to acknowledge the sponsors of our efforts, the ARPA Advance Simulation Technology Office (ASTO), in particular Lieutenant Colonel David Neyland, U. S. Air Force.

APPENDIX B: SAMPLE PROGRAM DATALOG.C

This appendix contains the listing of a sample program that uses the DIS network library. It demonstrates the use of the five basic user functions and selected memory management functions. The program is a data logger. It is used to record DIS PDUs from the network and transmitting the contents of a logged file. The user specifies whether the program will read from the network or write to the network in the command line. The user also specifies the name of the log file. The command line is:

datalog [-i (to read from a file) | -o (to read from the network)] <file name>

```
/* File:  datalog.c
* Description:This program reads DIS PDUs from network and
* writes them to a file /daily/zeswitz/<filename>. Filename
* is specified in the command line.It also reads the PDUs from
* a file and transmits them.
*
* Revision:1.0 - 15 Sep 93
*
* Author: Steve Zeswitz
*         CS Department, Naval Postgraduate School
*         Internet: zeswitz@taurus.cs.nps.navy.mil
*/
#include <unistd.h>
#include <sys/times.h>
#include <gl.h>
#include <device.h>
#include "disdefs.h"

#define LOG_PATH    "/daily/zeswitz/"
#define TOUCH       "touch "
#define HZ          100

main(argc, argv)
   int argc; char **argv;
{
    int             i = 0, print_help = FALSE,
                    read_flag,
                    write_flag,
                    nodes,
```

```
                PDUlength,
                eof = FALSE;
 char           *pdu, ether_interf[MAX_INTERF+1], c;
 PDUType        type;
 double         time_stamp, start_time;
 short          value;

 /* pointers to PDU structures */
 EntityStatePDU             *ESpdu;
 FirePDU                    *Fpdu;
 DetonationPDU              *Dpdu;
 ServiceRequestPDU          *SRpdu;
 ResupplyPDU                *Rpdu;
 ResupplyCancelPDU          *RCpdu;
 RepairCompletePDU          *RC_pdu;
 RepairResponsePDU          *RRpdu;
 CollisionPDU               *Cpdu;

 ArticulatParamsNode        *APNptr;
 SupplyQtyNode              *SQNptr;

 /* command line file info */
 char           DATALOG[40];        /* file name   */
 int            datalog;            /* file handle */
 char           touchfile[60];      /* cmd line    */

/* datalog file path */
strcpy(DATALOG,LOG_PATH);

/* Parse command line */
for (i = 1; i < argc; i++) {

  if ((argv[i][0]!= '-') || print_help) {
     print_help = TRUE;
     break;
  }

  switch (argv[i][1]) {
     case 'h':      /* help */
        print_help = TRUE;
        break;
     case 'o':      /* read PDUs and write to file */
        if (i+1 < argc) {
```

```
            read_flag = TRUE;
            write_flag = FALSE;
            strcat (DATALOG, argv[++i]);
          } else
             print_help = TRUE;
          break;
      case 'i':      /* read PDUs from file and transmit */
        if (i+1 < argc) {
           read_flag = FALSE;
           write_flag = TRUE;
           strcat (DATALOG, argv[++i]);
        } else
             print_help = TRUE;
        break;
      default:
        print_help = TRUE;
   }/* end switch */
}/* end for */

if (argc == 1 || print_help) {
   printf("Usage:%s [-o <file> to write to | -i <file> to
                  read from]|[-h]\n", argv[0]);
   exit(0);
}

strcpy(ether_interf, BCAST_INTERF);

strcpy(touchfile,TOUCH);
strcat(touchfile,DATALOG);
system(touchfile);

if (read_flag) {/* create datalog file */
   qdevice(ESCKEY);
   start_time = times(&timing_info);

   /********************/
   /* open the network */
   /********************/
   if (net_open(ether_interf) == FALSE) {
       printf("main(): net_open(\"%s\") failed\n",
                  ether_interf);
       exit(1);
   }
```

```
/* open the datalog file */
if ((datalog = open(DATALOG, O_WRONLY)) == -1) {
     perror("\nCould not open datafile\n");
     exit(0);
}

printf("************************************\n");
printf("  Logging file /daily/zeswitz/%s\n\n",argv[2]);
printf("  PRESS ESC KEY TO STOP \n");
printf("************************************\n");

while (TRUE) {

   if (qtest())  {
      if (qread(&value) == ESCKEY)
         break;
   }

    /*****************************/
    /* read from the network     */
    /*****************************/
   nodes = net_read(&pdu, &type);

   if (nodes == -1) {     /*Error reading the network*/
      printf("main(): Error on net_read()\n");
   } else if (nodes == 0) { /* No pending PDUs */
      continue;
   }

   /* time_stamp the PDU */
   time_stamp = times(&timing_info) - start_time;

   /* get the PDU length from the header */
   PDUlength = ((PDUHeader*) pdu)->length;

   /* write the data to the file */
   write(datalog, &time_stamp, sizeof(time_stamp));
   write(datalog, &type, sizeof(type));
   write(datalog, pdu, PDUlength);

   /* deallocate PDU */
   freePDU(pdu);
```

```
      }/* end while(TRUE) */

      /* close the file */
      close(DATALOG);
   }/* end if (read_flag) */

/*********************************************************/
/* read PDUs from datalog file and send to the network   */
/*********************************************************/
   if (write_flag) {/* write datalog file to the network */

      start_time = times(&timing_info);

      /*********************************/
      /* open the network to send only */
      /*********************************/
    if(net_open_select(ether_interf,0,(short *)0) == FALSE){
        printf("main():net_open(\"%s\") failed\n",
                    ether_interf);
        exit(1);
     }

     /* open the datalog file */
     if ((datalog = open(DATALOG, O_RDONLY)) == -1) {
         printf("\nCould not open datafile\n", DATALOG);
        fflush(stdout);
         exit(0);
     }

     do {
       /* read the time_stamp from the file */
      if (i = read(datalog, &time_stamp, sizeof(time_stamp))
                    == eof) {
        eof = TRUE;
        continue;
       }
       if ( i == -1) {
          perror("Error reading time_stamp from file");
          break;
       }
```

```
 /* read the PDU type from the file
 if (i = read(datalog, &type, sizeof(type)) == eof) {
    eof = TRUE;
    break;

 /* spin until it is time to send this pdu */
while ((times(&timing_info)-start_time)<time_stamp) {
 }

 switch (type) {

   case (CollisionPDU_Type):
     Cpdu=(CollisionPDU *)
              mallocPDU(CollisionPDU_Type);
     pdu = (char *) Cpdu;
     break;

   case (RepairResponsePDU_Type):
     RRpdu=(RepairResponsePDU *)
              mallocPDU(RepairResponsePDU_Type);
     pdu = (char *) RRpdu;
     break;

   case (RepairCompletePDU_Type):
     RC_pdu = (RepairCompletePDU *)
              mallocPDU(RepairCompletePDU_Type);
     pdu = (char *) RC_pdu;
     break;

   case (ResupplyCancelPDU_Type):
     RCpdu = (ResupplyCancelPDU *)
              mallocPDU(ResupplyCancelPDU_Type);
     pdu = (char *) RCpdu;
     break;

   case (ResupplyOfferPDU_Type):
   case (ResupplyReceivedPDU_Type):
     Rpdu = (ResupplyPDU *)
              mallocPDU(ResupplyOfferPDU_Type);
     pdu = (char *) Rpdu;
     break;

   case (ServiceRequestPDU_Type):
```

```
   SRpdu = (ServiceRequestPDU *)
            mallocPDU(ServiceRequestPDU_Type);
   pdu = (char *) SRpdu;
   break;

case (DetonationPDU_Type):
   Dpdu = (DetonationPDU *)
            mallocPDU(DetonationPDU_Type);
   if (i=read(datalog,Dpdu,DetonationPDUBaseLength)
            == eof) {
      eof = TRUE;
      continue;
   }
   if (i= -1)  {
      perror("Error reading PDU from file");
      exit(0);
   }
   pdu = (char *) Dpdu;
   break;

case (FirePDU_Type):
   Fpdu = (FirePDU *) mallocPDU(FirePDU_Type);
   if (i=read(datalog,Fpdu,sizeof(FirePDU)== eof) {
      eof = TRUE;
      continue;
   }
   if (i= -1)  {
      perror("Error reading PDU from file");
      exit(0);
   }
   pdu = (char *) Fpdu;
   break;

case (EntityStatePDU_Type):
   ESpdu = (EntityStatePDU *)
            mallocPDU(EntityStatePDU_Type);
   if(i=read(datalog,Dpdu,EntityStatePDUBaseLength)
           == eof) {
      eof = TRUE;
      continue;
   }
   if (i= -1)  {
      perror("Error reading PDU from file");
```

```
                exit(0);
            }
            pdu = (char *) ESpdu;
            break;

        default:
            printf("default case reached!\n");

    } /* end switch(type) */

    /*****************************/
    /* writing to the network    */
    /*****************************/
    if (net_write(pdu, type) == FALSE)
        sprintf(stderr, "net_write() failed\n");

    /* deallocate the PDU */
    freePDU(pdu);

    fflush(stdout);

  } while (!eof);

  /* close the file */
  close(DATALOG);

}/* end if (write_flag) */

 /*********************/
 /* close the network */
 /*********************/
 net_close();
 exit(0);
}

/* EOF */
```

APPENDIX C: DIS NETWORK LIBRARY MANUAL PAGES

This appendix contains DIS Network Library manual pages. It provides a quick reference for the purpose and use of key network library functions.

NAME

net_open-open the network to send and receive DIS protocol data units.

SYNOPSIS

```
#include <disdefs.h>

int net_open(interf)
    char *interf;
```

DESCRIPTION

net_open returns a value of 1 when the network is opened successfully. A value of zero is returned when opening the network fails. The argument, interf, is the name of the Ethernet hardware interface to be used for communication. It is only used when the system confirms its presence, otherwise the system provides the name of the interface and a message displays the name of the interface is being used.

SEE ALSO

net_open_select(), net_close()

AUTHOR

Steven R. Zeswitz

NAME

net_open_select-open the network to send DIS protocol data units and receive only requested protocol data units.

SYNOPSIS

```
#include <disdefs.h>

int net_open(interf, num, typelist)

    char    *interf;
    int     num;
    short   *typelist;
```

DESCRIPTION

net_open_select returns a value of 1 when the network is opened successfully. A value of zero is returned when opening the network fails. The argument, interf, is the name of the Ethernet hardware interface to be used for communication. It is only used when the system confirms its presence, otherwise the system provides the name of the interface and a message displays the name of the interface is being used. num specifies the number of PDU types to be included in the incoming PDU queue. typelist is an array of PDU types. The types included in the list are appended to the incoming queue.

SEE ALSO

net_open(), net_close()

AUTHOR

Steven R. Zeswitz

NAME

net_read-read a PDU from the queue of incoming PDUs.

SYNOPSIS

```
#include <disdefs.h>

int net_read(pdu,type)

    char      **pdu
    PDUType *type;
```

DESCRIPTION

net_read returns a value equivalent to the number of PDUs in the queue. A value of zero is returned when there are no pending PDUs. A value of -1 is returned when the read operation fails. The argument pdu is a pointer to a memory structured PDU that is returned. The argument type returns the DIS PDU type.

NOTE

The network processes do not deallocate the memory used by the PDU that is returned. The memory must be deallocated with freePDU(PDUType).

SEE ALSO

freePDU()

AUTHOR

Steven R. Zeswitz

NAME

net_write-transmit a DIS PDU.

SYNOPSIS

#include <disdefs.h>

int net_write(pdu, ptype)

```
char        *pdu;
PDUType ptype;
```

DESCRIPTION

net_write returns a value of 1 when the send operation is successful. A value of zero is returned when the send operation fails. The argument pdu is the memory structured PDU to be transmitted and the argument ptype is the DIS PDU type.

SEE ALSO

mallocPDU()

AUTHOR

Steven R. Zeswitz

NAME

net_close-terminate the network process.

SYNOPSIS

#include <disdefs.h>

void net_close()

DESCRIPTION

net_close closes the send and receive sockets and terminates the receiving process.

SEE ALSO

net_open(), net_open_select()

AUTHOR

Steven R. Zeswitz

NAME

mallocpdu-allocate memory for a DIS PDU.

SYNOPSIS

#include <disdefs.h>

char *mallocPDU(ptype)

PDUType ptype;

DESCRIPTION

mallocPDU returns a pointer to a buffer allocated for a DIS PDU. NULL is returned when the operation fails. The argument ptype is determines the size of the buffer.

SEE ALSO

net_write(), freePDU()

AUTHOR

Steven R. Zeswitz

NAME

freePDU-deallocate memory used by a DIS PDU.

SYNOPSIS

```
#include <disdefs.h>

void *freePDU(pdu)
    char *pdu;
```

DESCRIPTION

freePDU deallocates the memory used by pdu.

SEE ALSO

net_write(), mallocPDU(), net_read()

AUTHOR

Steven R. Zeswitz

LIST OF REFERENCES

[Bogg92] Boggs, David R., Mogul, Jeffrey C., Kent, Christopher A., *Measured Capacity of an Ethernet: Myths and Realities*, WRL Research Report 88/4, Western Research Laboratory, Palo Alto, California, September 1988.

[Blum92] Blumenthal, Steven, *Future Trends in Networking Technology for Distributed Simulation*, Presentation to the Defense Science Board Sub-Panel on Technology Forecast, BBN Systems, Cambridge, Massachusetts, July 1992.

[Chun92] Chung, James W., *An Assessment and Forecast of Commercial Enabling Technologies for Advanced Distributed Simulation*, Institute for Defense Analysis, Arlington, Virginia, October 1992.

[DoD92] Department of Defense, *Defense Modeling and Simulation Initiative*, Washington D. C., May 1992.

[Frie88] Friedman, Dan, Haimo, Varda, *SIMNET Ethernet Performance*, BBN Communications Corporation, Cambridge, Massachusetts, January 1988.

[Hamr93] Hamre, John, *Key Note Remarks to the Eighth Workshop on Interoperability of Defense Simulations*, Orlando, Florida, March 1993.

[Harv92] Harvey, Edward P., Schaffer, Richard L., *The Capability of the Distributed Interactive Simulation Networking Standard to Support High Fidelity Aircraft Simulation*, BMH Associates, Inc. and BBN Systems and Technologies, Norfolk VA, Cambridge, Massachusetts, July 1992.

[Hedr87] Hedrick, Charles L., *Introduction to Internet Protocols*, Rutgers University, September 1988.

[IEEE93] Institute of Electrical and Electronics Engineers, International Standard, ANSI/IEEE Std 1278-1993, *Standard for Information Technology, Protocols for Distributed Interactive Simulation*, March 1993.

[IEEE85] Institute of Electrical and Electronics Engineers, International Standard, ANSI/IEEE Std 802.3-1988, *Information Processing Systems, Local Area Networks, Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA-CD) Access Method and Physical Layer Specifications*, First Edition, December 1989.

[IST93] Institute for Simulation & Training, IST-TR-93-11, *Distributed Interactive Simulation Guidance Document [Draft 2.1]*, University of Central Florida, Orlando, Florida, March 1993.

[IST93a] Institute for Simulation & Training, IST-TR-93-11, *Distributed Interactive Simulation Operational Concept [Draft 2.2]*, University of Central Florida, Orlando, Florida, March 1993.

[IST93b] Institute for Simulation and Training, IST-TR-93-20, *Communication Architecture for Distributed Interactive Simulation (CADIS) [Final Draft]*, University of Central Florida, Orlando, Florida, June 1993.

[IST93c] Institute for Simulation and Training, IST-TR-93-20, *Guidance Document Communication Architecture for Distributed Interactive Simulation (CADIS) [Draft]*, University of Central Florida, Orlando, Florida, June 1993.

[IST93d] Institute for Simulation and Training, IST-TR-93-20, *Rationale Communication Architecture for Distributed Interactive Simulation (CADIS)*, University of Central Florida, Orlando, Florida, June 1993.

[IST93e] Institute for Simulation and Training, IST-CR-93-02, *Enumeration and Bit Encoded Values for Use with Protocols for Distributed Interactive Simulation Applications*, University of Central Florida, Orlando, Florida, March 1993.

[IST93f] Institute for Simulation and Training, IST-CR-93-15, *Standard for Information Technology, Protocols for Distributed Interactive Simulation Applications [Proposed IEEE Standard Draft]*, University of Central Florida, Orlando, Florida, June 1993.

[IST93g] Institute for Simulation and Training, IST-TR-93-08, *Simulator Networking Handbook*, University of Central Florida, Orlando, Florida, June 1993.

[Jaco92] Jacobsen, Van, Leres, Craig, McCanne, Steven, TCPDUMP, Lawrence Berkeley Laboratory, University of California, Berkeley, California,

[Lock] Locke, John, Pratt, David R., and Zyda, Michael J., *A DIS Network Library for UNIX and NPSNET*, Naval Postgraduate School, Monterey, California,, undated

[Lock92] Locke, John, Pratt, David R., and Zyda, Michael J., *Integrating SIMNET with NPSNET Using a Mix of Silicon Graphics and Sun Workstations*, Naval Postgraduate School, Monterey, California, March 1992.

[Lora92] Loral Systems Company, *Strawman Distributed Interactive Simulation Architecture Description Document Volume 1*, Advanced Distributed Simulation Technology Program Office, Orlando, Florida, March 1992.

[Pope89] Pope, Arthur, BBN Report No. 7102, *The SIMNET Network and Protocols*, BBN Systems and Technologies, Cambridge, Massachusetts, July, 1989.

[Prat93] Pratt, David R., *A Software Architecture for the Construction and Management of Real Time Virtual Worlds*, Dissertation, Naval Postgraduate School, Monterey, California, June 1993

[Redd92] Reddy, Bob, Col USA, *Advanced Distributed Simulation Concept Briefing*, Defense Advanced Research Projects Agency, November 1992.

[SGI91] Silicon Graphics, Inc., Document Number 007-0810-030,*IRIS Network Programming Guide*, Mountain View, CA, 1991.

[SRI92] SRI International, *ATD-1 Architecture White Paper Edit Draft*, Menlo Park, CA, undated.

[Stal91] Stallings, William, *Data and Computer Communications*, Third Edition,, Macmillan Publishing, 1991.

[Thor87] Thorpe, Jack A., LtCol USAF, *The New Technology of Large Scale Simulator Networking: Implications For Mastering the Art of Warfighting*, Defense Advanced Research Projects Agency, Arlington, Virginia, November 1987.

[Zyda92] Zyda, Michael J., Pratt, David R., Kelleher, Kristen M., *1992 NPSNET Research Group Overview*, Naval Postgraduate School, Monterey, California, May 1993.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
Dudley Knox Library Code 052 Naval Postgraduate School Monterey, CA 93943-5002	2
Dr. Ted Lewis, Chairman and Professor Computer Science Department Code CS/LT Naval Postgraduate School Monterey, CA 93943	1
Dr. David R. Pratt, Assistant Professor Computer Science Department Code CS/PR Naval Postgraduate School Monterey, CA 93943	2
Dr. Gilbert M. Lundy, Assistant Professor Computer Science Department Code CS/LN Naval Postgraduate School Monterey, CA 93943	2
Michael J. Zyda, Professor Computer Science Department Code CS/ZK Naval Postgraduate School Monterey, CA 93943	1
LtCol David L. Neyland USAF ARPA/ASTO 3701 Fairfax Drive Arlington, VA 22203	1
Maj Michael Macedonia USA Computer Science Department Code CS Naval Postgraduate School Monterey, CA 93943	1

LCdr Don Brutzman USN 1
Computer Science Department Code CS
Naval Postgraduate School
Monterey, CA 93943

Capt. Steven R. Zeswitz USMC 2
MCCDC
Quantico, VA 22134